版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

Broadview®
www.broadview.com.cn

全栈开发者倾情奉献，全方位讲解Go语言的核心内容，
从源码层面了解其实现原理。



Go语言

编程入门与实战技巧

黄靖钧◎编著

体系完整： 通过142个案例，囊括Go语言的主要功能
循序渐进： 从基本语法到网络编程，条理清晰，逐步提高
实用参考： 包括18个常用标准库，深度讲解Go语言的实战技巧
原理剖析： 深入理解Go语言源码实现，了解程序优化方法

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

仅供非商业用途或交流学习使用



作者简介

黄靖钧

全栈开发者，长期以来一直使用Go语言开发网络应用，在Go语言编程方面经验丰富。热衷开源，深度参与Docker及其生态建设，积极为portainer等众多著名项目贡献代码。曾做PaaS与CaaS等类型项目的后端Go语言开发工作，现从事基础设施架构与数据中心开发工作，专注高并发、微服务领域的研究。



仅供非商业用途或交流学习使用



Go语言

编程入门与实战技巧

黄靖钧◎编著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING



仅供非商业用途或交流学习使用



内 容 简 介

本书从内容上分为三大部分，第一部分主要介绍 Go 语言的基础知识，包括 Go 语言的安装和开发工具，介绍了 Go 语言的特性与适合的场景，然后讲解了 Go 语言的程序结构和数据类型，并针对函数和一些关键字的用法与数据类型的调用原理做了阐述。第二部分介绍了 Go 语言数据结构和标准库，结合实际应用场景探讨了日常生产环境会遇到的问题与解决办法。第三部分主要介绍 Go 语言的测试工具和用法，并重点讲解了 Go 语言的内存管理机制，深入理解 Go 语言的设计哲学，了解 Go 语言底层的内存管理和并发机制，为更进一步的学习打下坚实的基础。

本书适合对计算机编程尤其是对 Go 语言编程感兴趣的新手作为入门教程阅读，还适合想在 Web 开发领域有所发展的程序员学习。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

Go 语言编程入门与实战技巧 / 黄靖钧编著. —北京：电子工业出版社，2018.9

ISBN 978-7-121-34966-9

Ⅰ. ①G… Ⅱ. ①黄… Ⅲ. ①程序语言—程序设计Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字（2018）第 198989 号

策划编辑：张月萍
责任编辑：牛　勇　　特约编辑：赵树刚
印　　刷：三河市双峰印刷装订有限公司
装　　订：三河市双峰印刷装订有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　1/16　　印张：26.25　　字数：551 千字
版　　次：2018 年 9 月第 1 版
印　　次：2018 年 9 月第 1 次印刷
定　　价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件到 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。



仅供非商业用途或交流学习使用



前言

当你打开这本书时，想必对 Go 语言是有一定兴趣的，本书希望能够让你真正喜欢上 Go 语言。本书面向的读者是对 Go 语言有一定了解但尚未入门的新手。

Go 语言是谷歌公司发布的一款开源编程语言，它对多处理器系统应用程序的编程进行了大量优化。作为新时代的代表性新生编程语言，Go 语言素有云计算时代的 C 语言之称。从 2009 年正式发布至今，Go 语言经过频繁的迭代更新走到今天，两次斩获 TIOBE 编程语言排行榜的年度语言称号（2009 年和 2016 年），已经跻身主流编程语言之列。

时至今日，Go 语言编译程序的速度可以媲美 C 或 C++程序，而且更加安全，支持并行进程。整个 Go 语言社区在诸多前辈们的推广与布道下，不断壮大，形成了今天富有活力的生态环境。

Go 语言在最近几年的更新中有很多富有里程碑意义的更新，例如实现了自举，Go 语言几乎完全使用 Go 语言重写了，仅保留一些基础的汇编代码。实现自举的好处有很多，一是提高了执行速度，可以避免跨语言调用或虚拟机等情况；二是提高了安全性，缩小了程序攻击面，使得整个项目更加可控；三是能够更轻易地实现跨平台编译，编写跨平台程序变得轻而易举。

除此之外，Go 语言的运行时系统和标准库等大量生产工具都得到了极大的提升，每一次版本迭代，Go 语言都会通过社区反馈权衡内存管理机制，并对调度器做出一些调整。在即将开始的 Go 语言学习过程中，相信你也能够深刻感受到 Go 语言开发团队的深思熟虑。

Go 语言还有一堆令人“爱不释手”的工具，令人惊艳的 go tool 宛如一把“瑞士军刀”，只需安装 Go 语言环境之后就可以直接使用。



仅供非商业用途或交流学习使用



重要的是，Go 语言社区生态成绩同样斐然，随着社区扩大，Go 语言包管理机制也有了很大改进，vendor 目录的启用标志着 Go 语言开发进入了一个更加规范的阶段。

整个社区里，拿得出手的杀手级应用（如 Moby、Kubernetes、Gogs、Grafana、Etcd 等）喷涌而出。Go 语言累计接受了将近 4 万次的代码提交，近千名开发者参与到 Go 语言开发中。

本书的内容分为三大部分，第一部分主要介绍了 Go 语言的基础知识，包括 Go 语言的一些特性、适合的场景，以及 Go 语言的程序结构和数据类型，还讲解了函数和一些关键字的用法与数据类型的调用原理；第二部分介绍了 Go 语言常见的数据结构和常用的标准库，结合实际应用场景探讨了大部分生产环境会遇到的问题与解决办法；第三部分主要介绍了 Go 语言的测试与内存管理机制，深入理解 Go 语言的设计哲学，了解 Go 语言底层的内存管理和并发机制，为更进一步的学习打下坚实的基础。

本书的目标读者是对计算机编程尤其是对 Go 语言编程感兴趣的新手。由于作者水平和时间有限，书中难免会有一些错误和纰漏，欢迎读者指正。衷心希望通过本书的学习，能够让你对 Go 语言有一个比较全面的理解。



仅供非商业用途或交流学习使用

目录

第 1 章　认识 Go 语言 1

1.1　Go 语言简介 1

1.1.1　Go 语言简史 1

1.1.2　Go 语言特性 2

1.1.3　谁适合使用 Go 语言 2

1.2　Go 语言的开发环境部署 3

1.2.1　Go 语言环境变量 3

1.2.2　在 Linux 上安装 Go 语言环境 3

1.2.3　在 Mac OS 上安装 Go 语言环境 5

1.2.4　在 Windows 上安装 Go 语言环境 5

1.3　Go 语言的基本命令及使用 5

1.3.1　GOROOT 结构 6

1.3.2　GOPATH 结构 7

1.3.3　Go 语言命令行参数 8

1.3.4　第一个 Go 语言程序 12

1.4　Go 语言的开发工具 14

1.4.1　Code/Atom 14

1.4.2　VIM-Go 17

1.4.3　Gogland 18

1.4.4　LiteIDE 18

1.4.5　Cloud IDE 19

1.5　本章小结 23

第 2 章　程序结构 24

2.1　Go 语言程序元素 24

仅供非商业用途或交流学习使用



2.1.1 标识符.....24
2.1.2 关键字.....25
2.1.3 字面量.....25
2.1.4 分隔符.....25
2.1.5 运算符.....26
2.1.6 注释.....27
2.2 Go 语言基本概念.....28
2.2.1 常量.....28
2.2.2 变量.....32
2.3 本章小结.....37
第 3 章 基础数据类型.....38
3.1 整型.....39
3.1.1 整型的表示.....39
3.1.2 整型的运算.....39
3.2 浮点型.....41
3.2.1 浮点型的表示.....41
3.2.2 浮点型的运算.....42
3.2.3 复数.....42
3.3 字符与字符串.....44
3.3.1 字符串的表示.....44
3.3.2 操作字符串.....46
3.3.3 字符串格式化.....60
3.4 布尔型.....61
3.4.1 布尔型的表示.....61
3.4.2 布尔型的运算.....62
3.5 基本数据类型的扩展.....63
3.5.1 强制类型转换.....63
3.5.2 自定义类型.....64
3.5.3 类型别名.....64
3.5.4 指针.....65
3.6 本章小结.....69
第 4 章 流程控制.....70
4.1 条件语句.....70



仅供非商业用途或交流学习使用



4.1.1 if 判断 71
4.1.2 if-else 判断 71
4.1.3 else-if 判断 73
4.1.4 初始化子语句 74
4.2 选择语句 75
4.2.1 switch 语句 75
4.2.2 switch 初始化语句 78
4.2.3 select 语句 79
4.3 循环语句 80
4.3.1 for 的子语句 81
4.3.2 range 子语句 82
4.4 延迟语句 84
4.5 标签 87
4.5.1 break 87
4.5.2 continue 89
4.5.3 goto 90
4.6 本章小结 91
第 5 章 函数 92
5.1 认识函数 92
5.1.1 函数的声明 92
5.1.2 函数的参数 93
5.1.3 函数的返回值 94
5.2 函数的基础 95
5.2.1 多返回值 96
5.2.2 函数作为参数 97
5.2.3 函数作为类型 98
5.2.4 可变参数 99
5.2.5 匿名函数与闭包 102
5.2.6 递归函数 106
5.2.7 内置函数 108
5.3 函数进阶 108
5.3.1 参数传递机制 108
5.3.2 defer 与跟踪 111
5.3.3 错误与恢复 116
5.4 本章小结 122



第 6 章　复合数据类型 123
6.1　数组 123
6.1.1　声明数组 123
6.1.2　访问与修改 124
6.1.3　多维数组 126
6.1.4　将数组传递给函数 127
6.2　切片 128
6.2.1　创建数组切片 129
6.2.2　切片的使用 131
6.2.3　多维切片 137
6.2.4　将切片传递给函数 137
6.3　映射 138
6.3.1　映射的实现 138
6.3.2　映射的创建 139
6.3.3　映射的使用 139
6.3.4　将映射传递给函数 141
6.4　本章小结 142
第 7 章　包 144
7.1　包的基础 144
7.1.1　包的结构 146
7.1.2　包的导入 147
7.1.3　包的使用 153
7.1.4　Go 语言工具箱 154
7.2　自定义包 156
7.2.1　包的制作 157
7.2.2　特定平台的代码 157
7.2.3　godoc 生成文档 158
7.2.4　包的打包与发布 159
7.2.5　自定义包的导入 164
7.3　本章小结 165
第 8 章　结构体与方法 166
8.1　结构体 166
8.1.1　结构体定义 166

8.1.2 结构体使用 172
8.1.3 带标签的结构体 175
8.1.4 匿名字段和内嵌结构体 176
8.2 类型系统 179
8.2.1 用户自定义类型 179
8.2.2 值语义和引用语义 182
8.3 方法 184
8.3.1 方法声明 184
8.3.2 为类型添加方法 187
8.3.3 工厂方法创建结构体 188
8.3.4 基于指针对象的方法 190
8.3.5 方法值和方法表达式 191
8.3.6 方法和未导出字段 193
8.3.7 嵌入类型的方法和继承 194
8.4 本章小结 197
第 9 章 接口与反射 198
9.1 接口 198
9.1.1 接口是什么 199
9.1.2 接口类型与约定 204
9.1.3 接口实现 208
9.1.4 嵌套接口 210
9.1.5 接口赋值 211
9.1.6 接口查询 213
9.1.7 接口组合 214
9.2 反射 215
9.2.1 方法和类型的反射 215
9.2.2 通过反射修改设置值 216
9.2.3 反射结构 218
9.2.4 Printf 和反射 220
9.3 本章小结 222
第 10 章 并发编程 223
10.1 并发编程基础 223
10.1.1 并发与并行 223

10.1.2 指定使用核心数 225
10.2 协程（goroutine） 227
10.2.1 协程基础 228
10.2.2 协程间通信 229
10.3 通道（channel） 231
10.3.1 基本语法 232
10.3.2 select 232
10.3.3 缓冲机制 233
10.3.4 超时和计时器 234
10.3.5 channel 的传递 235
10.3.6 单向 channel 236
10.3.7 关闭 channel 237
10.4 并发进阶 237
10.4.1 多核并行化 237
10.4.2 协程同步 239
10.4.3 协程和恢复 242
10.5 本章小结 243
第 11 章 网络编程 244
11.1 Socket 编程 244
11.1.1 什么是 Socket 244
11.1.2 Dial()函数 246
11.1.3 ICMP 示例 247
11.1.4 TCP Socket 249
11.1.5 UDP Socket 253
11.2 HTTP 编程 255
11.2.1 HTTP 客户端 255
11.2.2 HTTP 服务器端 260
11.3 RPC 编程 262
11.3.1 Go RPC 263
11.3.2 HTTP RPC 263
11.3.3 TCP RPC 266
11.3.4 JSON RPC 268
11.3.5 RPC 接口 271
11.4 数据库 272

11.4.1 database/sql 接口 272
11.4.2 使用 MySQL 数据库 277
11.4.3 使用 SQLite 数据库 280
11.4.4 使用 PostgreSQL 数据库 282
11.4.5 NoSQL 数据库操作 286
11.5 Go 语言使用 Cookie 287
11.5.1 设置 Cookie 287
11.5.2 读取 Cookie 288
11.6 本章小结 288
第 12 章 I/O 编程 289
12.1 输入/输出 289
12.1.1 io：基本 I/O 接口 289
12.1.2 fmt：格式化 I/O 292
12.1.3 文本处理 298
12.2 文件系统 298
12.2.1 os：系统功能实现 298
12.2.2 path：兼容路径操作 299
12.3 数据结构与算法 303
12.3.1 排序 304
12.3.2 container 310
12.4 本章小结 314
第 13 章 文件处理 315
13.1 文件操作 315
13.1.1 创建文件与查看状态 316
13.1.2 重命名与移动 319
13.1.3 打开与关闭 319
13.1.4 删除与截断 321
13.1.5 读写文件 321
13.1.6 权限控制 325
13.1.7 文件链接 328
13.2 XML 处理 330
13.2.1 解析 XML 330
13.2.2 生成 XML 333

13.3 JSON 处理 336
13.3.1 解析 JSON 336
13.3.2 生成 JSON 338
13.4 日志记录 340
13.4.1 Logrus 341
13.4.2 Seelog 342
13.5 压缩 343
13.5.1 打包与解包 343
13.5.2 压缩与解压 345
13.6 本章小结 347
第 14 章 安全与测试 348
14.1 安全 348
14.1.1 安全相关的基础概念 348
14.1.2 通信安全 350
14.2 测试 354
14.2.1 单元测试 354
14.2.2 基准测试 362
14.3 本章小结 368
第 15 章 内存管理 369
15.1 内存分配 369
15.1.1 内存管理基本概念 369
15.1.2 逃逸分析 373
15.2 TCMalloc 376
15.2.1 整体结构 377
15.2.2 小内存分配 378
15.2.3 CentralCache 379
15.2.4 大内存分配 381
15.3 Mspan（内存管理器） 384
15.4 垃圾回收 398
15.4.1 标记清理算法 399
15.4.2 标记实现 403
15.4.3 清理 406
15.4.4 监控 408
15.5 本章小结 408

第 1 章
认识 Go 语言

本书开篇，先来了解 Go 语言的一些特性、历史以及优缺点；接下来根据不同系统介绍安装 Go 语言环境的方法，了解 Go 语言的基本命令以及开发 Go 语言的常用工具。

1.1 Go 语言简介

Go 语言是 Google 于 2009 年正式发布的一款开源的静态编译型编程语言。

1.1.1 Go 语言简史

Go 语言最早于 2007 年由 Rob Pike（贝尔实验室 UNIX 团队成员，曾参与 Plan 9、Inferno 和 Limbo 等项目）、Robert Griesemer（Java HotSpot 虚拟机、V8 引擎开发者之一）和 Ken Thompson（贝尔实验室 UNIX 团队成员，C 语言、UNIX 和 Plan 9 创始人之一，与 Rob Pike 共同开发了 UTF-8 字符集规范）三人在业余时间联合开发，后来还加入了 Ian Lance Taylor（GCC 核心开发人员）、Russ Cox（曾参与 Plan 9 操作系统的开发）和 Brad Fitzpatrick（memcached 的作者）等人。通过这几个人的履历就可以知道，Go 语言的创始团队对操作系统和系统编程语言有着非常深刻的理解，这对一门编程语言的开发起到至关重要的作用。

Google 于 2009 年 11 月 10 日正式发布 Go 语言，并以 BSD 协议完全开源，支持 Linux 和 Mac OS 平台，同年 11 月支持 Windows 平台。时至今日，Go 语言已经完成自举，社区生态成绩斐然，包括大量拿得出手的杀手级应用（Moby、Kubernetes、Gogs、Grafana、Etcd 等），两次夺得 TIOBE 年度编程语言称号（2009 年和 2016 年）。Go 语言累计接受了 3 万多次代码提交，共有近 900 名开发者参与到 Go 语言项目中。

1.1.2 Go 语言特性

Go 语言作为一门静态类型的编译型语言，与当前的传统开发语言（如 Java、PHP）相比具备许多新特性。例如 Go 语言拥有自动垃圾回收功能，同时也允许开发人员干预回收操作；Go 语言有着更加丰富的内置类型，在错误处理方面语法更加精简高效。在 Go 语言中，函数支持多个返回值，而且函数也是一种值类型，可以作为参数传递。

从 struct 关键字就可以看出，Go 语言的类型定义参考了 C 语言中的结构（struct），但是 Go 语言并不像 C++和 Java 那样设计一个庞杂的类型系统，而是仅支持最基本的类型组合，不支持继承和重载。虽然 Go 语言没有类和继承的概念，但是它通过接口（interface）的概念来实现多态性。

Go 语言对多核处理器的编程进行了优化，Go 语言从程序与结构方面来实现并发编程，这是 Go 语言最重要的特性之一。

Go 语言和其他语言一样，拥有一个健全的包管理机制，同时得益于包之间的树状依赖，Go 语言的初次编译速度媲美 C/C++，甚至二次编译的速度明显快于 C/C++，同时又拥有接近 Python 等解释语言的简洁和开发效率。Go 语言在执行速度、编译速度和开发效率之间做了权衡，尽量达到了快速编译，高效执行，易于开发的目标。

最后，Go 语言支持交叉编译，可以在运行 Linux 系统的计算机上开发 Windows 下的应用程序。Go 语言源码文件格式默认都是使用 UTF-8 编码的。

1.1.3 谁适合使用 Go 语言

编程语言说到底是一种工具，不选最好的，只选最适合的，那么 Go 语言适合哪些场景?

首先使用 Go 语言，可以让 Web 服务器端的开发变得更高效，能够充分发挥多核计算机的性能，拥有更出色的网络环境兼容能力。自动垃圾回收、类型安全、依赖严格、编译快速等特点都是 Go 语言的魅力所在。很显然，Go 语言的目标就是针对服务器端的 Web 开发领域。

Go 语言凭借其出色的并发能力，在高性能分布式系统领域如鱼得水，像集群系统、游戏服务器端等场景都可以把 Go 语言作为首选开发语言。但是，Go 语言并不适合开发强实时性的软件，垃圾回收和自动内存分配等因素导致 Go 语言在实时性上有些力不从心。

1.2　Go 语言的开发环境部署

目前 Go 语言支持 Linux、FreeBSD、Mac OS 和 Windows 平台，安装包可以从 https://golang.org/dl 下载。

1.2.1　Go 语言环境变量

与 Java 等编程语言一样，安装 Go 语言开发环境需要设置全局的操作系统环境变量（除非使用包管理工具直接安装）。

主要的系统级别的环境变量有如下这些。

- $GOROOT：表示 Go 语言环境在计算机上的安装位置，它的值可以是任意你喜欢的位置（方便管理为原则），这个变量只有一个值，值的内容必须是绝对路径。
- $GOPATH：这是 Go 语言的工作目录，可以有多个，类似于工作空间的概念。一般不建议将$GOPATH 和$GOROOT 设置为同一个目录。

1.2.2　在 Linux 上安装 Go 语言环境

首先下载 Linux 的安装包，如图 1-1 所示，选择 Linux 安装包。

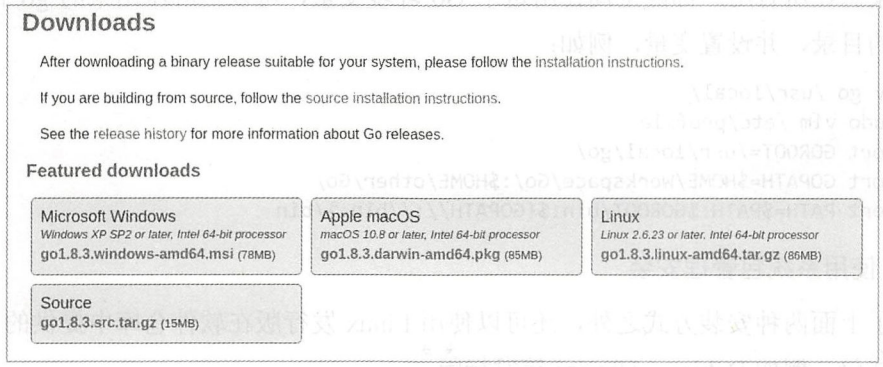


图 1-1　下载界面

下载之后，本地解压：

```
$ tar xf go1.8.3.linux-amd64.tar.gz
```

1. 当前用户

解压之后可以看到一个名为 go 的文件夹，把这个文件夹移动到你喜欢的方便管理的位置，然后编辑$HOME/.profile 文件：

```
$ vim $HOME/.profile
```

在文件末尾加入如下内容：

```
export GOROOT=$HOME/Applications/go/
export GOPATH=$HOME/Workspace/Go/:$HOME/other/Go/
export PATH=$PATH:$GOROOT/bin:${GOPATH//://bin:}/bin
```

上面的例子中，把 Go 语言环境放在$HOME/Applications/go/目录中，同时把$HOME/Workspace/Go/和$HOME/other/Go/设置为 Go 语言的工作目录。虽然$GOPATH 可以设置多个，但以后包管理安装时默认使用第一个$GOPATH 的值作为下载目录，建议把$GOPATH 第一个值设置为常用的全局工作目录，例如以后安装一些 Go 语言工具时，它们的 bin 执行文件可以在这里找到。

第三句是把前两个变量输出到系统$PATH 变量中，这样在当前用户环境中就可以随时使用 Go 语言环境了。

2. 所有用户

如果想让所有用户都能全局使用这个 Go 环境变量，可以把解压出来的 go 目录移动到相应的目录，并设置变量，例如：

```
$ mv go /usr/local/
$ sudo vim /etc/profile
export GOROOT=/usr/local/go/
export GOPATH=$HOME/Workspace/Go/:$HOME/other/Go/
export PATH=$PATH:$GOROOT/bin:${GOPATH//://bin:}/bin
```

3. 使用系统包管理安装

除了上面两种安装方式之外，还可以使用 Linux 发行版在软件仓库中提供的 Go 语言环境安装包，例如 Debian、Ubuntu 等发行版：

```
$ sudo apt install golang-go
```

其他发行版类似，一般软件包的名称不是 golang 就是 go，或者组合词。

1.2.3　在 Mac OS 上安装 Go 语言环境

同样从如图 1-1 所示的页面可以下载 Mac OS 版本的安装包，双击鼠标进行安装，根据引导安装完成即可，默认安装到/usr/local/go/目录。如果之前已经安装 Go 语言环境，需要卸载原来的版本再安装，即删除/etc/paths.d/go 文件。

Mac OS 设置变量与 Linux 一样，安装结束之后使用命令查看 Go 语言是否已经安装成功：

```
$ go version
```

1.2.4　在 Windows 上安装 Go 语言环境

同样在如图 1-1 所示的页面下载 Windows 版本安装包，下载的文件是一个可执行文件，直接双击进行安装。默认安装位置是 C:\Go 目录。

然后配置环境变量，Windows 同样有针对所有用户的系统变量与针对当前用户的用户变量，这里以设置系统变量为例，针对所有用户都有效。

用鼠标右键单击“我的电脑”，从弹出的快捷菜单中选择“属性”会打开系统控制面板，然后在左侧单击“高级系统设置”，在弹出的对话框中单击“环境变量”按钮，打开“环境变量”对话框，上半部分是用户变量配置，下半部分是系统变量配置。

在“系统变量”中单击“新建”按钮，输入“变量名”GOROOT，“变量值”是刚刚安装的 Go 语言路径 C:\Go，这样就配置好 Go 语言的安装路径了。

最后修改 PATH 系统变量，在变量值最后添加%%GOROOT\bin 路径，与其他 PATH 变量以分号“;”分隔。重启 CMD，在 CMD 中输入 go version 查看是否正常输出版本号。

1.3　Go 语言的基本命令及使用

安装 Go 语言环境之后就可以使用 go 命令了，直接输入 go help 查看有哪些命令可用：

```
$ go help
```

go 是用于管理 Go 语言源代码的工具。

用法：

```
go command [arguments]
```

包含的命令如下。

- build：编译源代码包和依赖。

- clean：删除对象文件。
- doc：显示 Go 包或程序实体的文档。
- env：打印输出 Go 语言环境的变量信息。
- bug：bug 提交程序。
- fix：修复程序。
- fmt：格式化源码包中的代码。
- generate：通过扫描 Go 源码中的 go:generate 注释来识别要运行的常规命令。
- get：下载并安装指定的包与依赖。
- install：编译并安装指定的包与依赖。
- list：打印指定源码包的信息。
- run：编译并运行 Go 程序。
- test：测试一个源码包。
- tool：运行一个指定的 go tool。
- version：打印输出 Go 环境版本。
- vet：检查源码包中可能出现的错误。

使用"go help [command]"查看每一个命令的使用细节。

这些 Go 语言自带的命令实际上来自$GOROOT/bin 与$GOROOT/pkg/tool/linux_amd64/目录下的可执行文件。1.3.3 节会详细介绍 Go 语言常用的基础命令。在此之前先来看 GOROOT 与 GOPATH 目录中有哪些需要注意的地方。

1.3.1 GOROOT 结构

$GOROOT 作为 Go 语言环境的根目录，放置了下面这些内容：

```
$ cd $GOROOT && ls -l
drwxr-xr-x .... .... api
-rw-r--r-- .... .... AUTHORS # 所有参与 Go 语言开发的人员名单
drwxr-xr-x .... .... bin
drwxr-xr-x .... .... blog # Go 官方博客的所有资源
-rw-r--r-- .... .... CONTRIBUTING.md # 为 Go 语言贡献代码的说明
-rw-r--r-- .... .... CONTRIBUTORS # 所有贡献者名单
drwxr-xr-x .... .... doc
-rw-r--r-- .... .... favicon.ico # Go 语言的图标
drwxr-xr-x .... .... lib # 存放一些特殊的库文件
-rw-r--r-- .... .... LICENSE # Go 语言的开源协议
drwxr-xr-x .... .... misc # 辅助工具和说明
-rw-r--r-- .... .... PATENTS # Go 语言专利说明
```

```
drwxr-xr-x .... .... pkg
-rw-r--r-- .... .... README.md # 说明文件
-rw-r--r-- .... .... robots.txt # 禁止搜索引擎索引本地启动的 Go 文档
drwxr-xr-x .... .... src
drwxr-xr-x .... .... test # 存放测试验证相关的文件
-rw-r--r-- .... .... VERSION # Go 语言版本文件
```

- api 文件夹存放了包含公开的变量、常量、函数等 API 的列表，用于 Go 语言 API 检索。
- bin 文件夹主要用于存储标准命令执行文件，例如默认的 go、godoc、gofmt 三件套。
- doc 文件夹存放了标准库的文档，使用 godoc -http=:6060 命令可以启动文档服务，通过本地 6060 端口即可访问。
- pkg 文件夹用于存放编译 Go 语言标准库生成的文件，这个文件夹里面的文件根据不同系统与架构会有所不同，在$GOROOT 这个文件中的 pkg 一般只有标准库的编译缓存文件，这些文件编译 Go 程序时起到很大作用，例如 pkg/tool/linux_amd64/ 目录下就有很多 go tool 命令的执行文件。
- src 文件夹存放的是 Go 语言自己的源代码（包括标准库），阅读源码时就是看这里面的内容。

1.3.2　GOPATH 结构

上面说了$GOROOT 目录的结构和目录用途，接下来还有一个类似的目录就是$GOPATH，如果把$GOROOT 看作安装目录，$GOPATH 就是一个工作目录。我们开发 Go 语言程序时就是在这里进行的。在这个目录中，一般有 3 个文件夹，如下所示。

- bin：存放 go install 生成的可执行文件，前面把$GOPATH/bin 路径加入 PATH 环境变量后就可以在全局任意位置使用这个文件夹内的执行文件，就与前面配置的 $GOROOT/bin 一样，可以直接在终端里使用这些程序了。
- pkg：存放 go 编译生成的文件。
- src：存放我们开发的 Go 项目的源代码，不同工程项目的代码以包名区分。

下面来看 src 的目录结构：

```
$ tree -L 2 src
├── github.com
│   ├── acroca
│   ├── Bowery
│   ├── cweill
│   ├── derekparker
```

```
|   ├── fatih
|   ├── golang
|   ├── gopherds
... ...

├── golang.org
|   └── x
├── gopkg.in
|   └── qml.v1
├── sourcegraph.com
|   └── sqs
... ...
```

以上面为例，可以看到 Go 的包名或项目名称习惯以 domain/project 为格式进行命名，这与 Go 语言的 get 命令——Go 语言的包管理机制有着莫大关系。虽然这并不是强制规定，但是为了方便管理，建议也使用这种结构建立自己的本地项目目录。

$GOPATH 可以有很多个，如果有两个 Go 语言开发工作区，可以使用冒号分隔两个变量路径，但是第一个$GOPATH 会被作为默认的 go get 存放目录。

1.3.3 Go 语言命令行参数

1. 程序编译

前面已经介绍了 go build 命令用于编译指定的代码包以及它们的依赖包。如果执行 go build 命令时不后跟任何代码包，那么命令将试图编译当前目录的代码包。例如，编译代码包 hello，其中一个方法是进入 hello 目录并直接执行该命令：

```
$ ls
hello
$ cd hello && go build
```

以上述这种方式编译时，如果代码包 hello 是 example 项目（example 项目路径是一个工作区）中的一个代码包，这个 hello 包只有库源码文件和测试源码文件，那么编译之后不会生成任何文件。这就涉及 Go 语言的源代码文件分类了。

在 Go 语言中，源代码文件分为 3 种，分别是命令源代码文件、库源代码文件和测试源代码文件。它们不仅功能各不相同，写法也各有特点。命令源代码文件作为可执行程序的入口；库源代码文件用于集中放置各种程序实体（全局常量、全局变量、接口、结构体、函数等）；而测试源代码文件用于对前两种源代码文件中的程序实体的功能和性能进行测试，或者展现前两者中程序的使用方法。

所以上面编译一个库包是不能生成任何文件的，对于命令源代码文件，其实还可以指定单个文件编译：

```
$ cat hello.go
package main
import "fmt"
func main() {
        fmt.Println("Hello world! 你好，世界！")
}
$ go build hello.go
$ ls
hello  hello.go
$  ./hello
Hello world! 你好，世界！
```

上面就是我们的第一个 Go 语言程序（先不管这个程序的结构），此时目录中已有了一个可执行文件 01_hello，执行 go install 会把这个可执行文件移动到$GOPATH/bin 目录中，使得这个 01_hello 成为全局命令，要删除这个命令只需进入$GOPATH/bin 中删除它即可。

这种单文件程序还有一个更简单的运行方式：

```
$ go run hello.go
Hello world! 你好，世界！
$ ls
hello.go
```

可以看到，使用 go run 命令并不会保存可执行文件，而是直接执行输出结果。

关于这几个命令也有它们自己的选项参数，其中常用的几个选项如下。

- -a：这个选项用于强制重新编译所有相关的 Go 语言源代码包，包括标准库中的包。默认 Go 语言的依赖包如果已经构建过了，第二次就不会再构建，但如果需要对底层源码做出修改，就需要使用这个选项从头编译一次。
- -n：用于检查执行命令过程中实际会用到的命令，这个选项只输出要执行的命令，实际上并不会执行。
- -p：构建或测试时指定并行运行的程序数量，默认值是可用的 CPU 数量。
- -race：检查数据竞争问题，并发编程中会用到，只支持 AMD 64 架构。
- -v：打印被编译的包。
- -work：指定编译缓存工作目录，Linux 系统默认在/tmp 文件夹中。指定工作目录时，编译结束后不会自动删除缓存目录。
- -x：与上面的-n 类似，打印编译过程中执行的命令，但这个选项会执行这些编译命令。

除了上面这些选项，还有其他的选项可以使用--help 或者 go help 查看。使用 go tool 可以查看 Go 语言默认携带的一些工具：

```
$ go tool
addr2line、asm、cgo、compile、cover、dist、doc、fix、link、nm、objdump、pack、pprof、
tour、trace、vet
```

这些命令可以使用 go tool COMMAND -h 来查看它们各自的帮助信息，也可以在下面这些地址找到相应的解释：

官方文档：https://golang.org/cmd

中文文档（部分）：https://github.com/hyper0x/go_command_tutorial

编译之后如果需要清除当前源码包和关联源码包里编译生成的文件，可以使用 go clean 快速清理，添加-r 选项可以清理导入代码包中的构建缓存。

2. Go 语言包安装

如果使用过 Go 语言开发的工具，通常会看到介绍中有一个快速安装的命令是 go get，这是一个 Go 自带的包获取工具，使用它可以方便地从互联网拉取各种代码包。例如下面的例子，使用 go get 获取来自 GitHub 的某个代码包：

```
$ go get github.com/example/go-code
```

执行这句命令之后，可以在$GOPATH 的第一个路径的 src/github.com 目录中找到这个代码包的源代码。这就是为什么前面强烈建议读者使用域名加项目名称组合来创建项目的原因所在，这样的目录命名一是可以分割项目，不容易重名；二是能够迅速找到原项目的地址；三是能够方便获取代码包。

实际上 go get 命令不仅可以从 GitHub 下载代码包，还可以从其他代码版本控制系统（英文为 Version Control System，简称为 VCS）中远程导入代码包。目前 go get 命令所支持的 VCS 的信息如表 1-1 所示。

表 1-1　go get 命令支持的 VCS 列表

名　称	主命令	说　明
Mercurial	hg	Mercurial 是一款采用 Python 语言开发的轻量级分布式版本控制系统
Git	git	Git 是当前使用最广泛的开源分布式版本控制软件。主流的几家代码托管网站都支持使用 Git 作为版本控制软件
Subversion	svn	Subversion 是一款历史悠久的版本控制系统，它是第一个在版本控制系统中加入分支概念的分布式版本控制系统
Bazaar	bzr	Bazaar 同样是开源的分布式版本控制系统，例如 Launchpad 就是使用 Bazaar 的著名网站之一

具体的 Go 语言包内容由后面单独的一章来介绍。这里我们继续关注 go get 本身，来看看它的选项与参数：

```
$ go get -h
```

使用这条命令可以查看帮助信息，下面是几个主要的选项解释。

- -d：只执行下载动作，不执行安装动作。
- -f：不检查已下载代码包的导入路径，需要与-u 选项配合使用。
- -fix：下载代码包后先执行 fix 动作（修复代码兼容问题），然后再进行编译和安装。
- -insecure：允许 get 命令使用不安全的 HTTP 协议下载代码包。
- -t：让 get 命令同时下载安装指定的代码包的测试源码文件中的依赖代码包。
- -u：更新已有代码包与依赖包。在默认情况下，get 命令不会下载本地已经存在的代码包，需要使用此选项更新代码包。
- -v：打印要下载安装的代码包名称。

go get 同样可以使用-x 选项显示下载安装要执行的具体命令。

3．代码格式化

Go 语言有一个特色就是强制格式化代码，在你保存代码之后，可以使用 go fmt 工具格式化。当然即便没有手动执行格式化，在编译时，Go 语言编译器也会自动格式化所有源代码。

这种强制格式化代码的做法可以有效统一编码风格，在大型项目中这一点至关重要，方便阅读，可以迅速定位代码。

```
$ cat hello.go
package main
import "fmt"
func main() { fmt.Println("Hello world!你好，世界！") }
$ go fmt hello.go
$ cat hello.go
package main

import "fmt"

func main() {
        fmt.Println("Hello world! 你好，世界！")
}
```

实际上 go fmt 调用的是 gofmt 工具，使用-h 选项可以查看 gofmt 的帮助信息。

- -l：显示需要格式化的文件。
- -w：不将格式化结果打印到标准输出，而是直接保存到文件中。
- -r：添加形如“<原始内容> -> <替换内容>”的重写规则，方便批量替换。
- -s：简化文件中的代码。
- -d：显示格式化前后的不同（不写入文件），默认是 false。
- -e：打印所有的语法错误。默认打印每行的前 10 个错误。
- -cpuprofile：支持调试模式，将相应的 cpufile 写入指定的文件。

替换例子：

```
# 替换没有意义的括号
$ gofmt -r "(a) -> a" -w *.go
# 去除多余的 len(a)
$ gofmt -r "a[n:len(a)] -> a[n:]" -w *.go
# 调换两个参数位置
$ gofmt -r 'A.Func1(a,b) -> A.Func2(b,a)' -w *.go
```

1.3.4 第一个 Go 语言程序

现在可以编写本书的第一个程序了，创建一个组织目录，例如 code.local，然后建立一个项目名为 hello-world，最终目录路径为$GOPATH/src/code.local/hello-world/，接下来使用编辑器新建一个文件名为 main.go，内容如下：

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println("Hello World")
}
```

直接在 main.go 目录中运行 go run main.go 即可，也可以使用 go build 构建但不执行。下面解释这段代码的含义。

上面程序分为三个部分，第一部分是包归属，即 package main，package 是 Go 语言的一个关键字，用于定义当前代码所属的包，与 Java 里的 package 类似，作为模块化的标识。main 包是一个特殊的包名，它表示当前是一个可执行程序，而不是一个库。

第二部分就是 import，我们称为包的导入，import 也是一个 Go 语言关键字，表示要

导入的代码包，与 Java 的 import 关键字类似，导入后就可以使用，Go 语言要求只有用到的包才能导入，如果导入一个代码包又不使用，那么编译时会报错。fmt 是这个程序导入的一个包的包名，fmt 是标准库中的一个包，导入之后，下面就可以使用它的函数了。

第三部分就是程序的主体，func 是一个函数定义关键字，main 是程序主函数，表示程序执行的入口，fmt.Println 是 fmt 包里的函数，调用方法与其他语言类似，这个程序用于输出一段文字。Go 语言默认不需要分号结束每行代码，如果两行代码写在一行时才需要使用分号分隔，但是不建议这样书写代码。

编译之后使用 go install 就可以将这个简单的 Go 语言程序安装到本地$PATH 中，这样这个程序就可以全局使用了。

前面说过可以在 Linux 下开发 Windows 的程序，这里以 Hello World 程序为例，在 Linux 下构建 Windows 的程序。使用 go env 可以查看当前系统的 Go 语言环境变量：

```
GOARCH="amd64"
GOBIN=""
GOEXE=""
GOHOSTARCH="amd64"
GOHOSTOS="linux"
GOOS="linux"
GOPATH="/home/zuolan/Workspace/Go/"
GORACE=""
GOROOT="/home/zuolan/Applications/go"
GOTOOLDIR="/home/zuolan/Applications/go/pkg/tool/linux_amd64"
GCCGO="gccgo"
CC="gcc"
GOGCCFLAGS="-fPIC -m64 -pthread -fmessage-length=0 -fdebug-prefix-map=/tmp/go-build
231441723=/tmp/go-build -gno-record-gcc-switches"
CXX="g++"
CGO_ENABLED="1"
PKG_CONFIG="pkg-config"
CGO_CFLAGS="-g -O2"
CGO_CPPFLAGS=""
CGO_CXXFLAGS="-g -O2"
CGO_FFLAGS="-g -O2"
CGO_LDFLAGS="-g -O2"
```

注意两个重要的环境变量：$GOOS 和$GOARCH。其中$GOOS 指的是目标操作系统，它目前支持 10 种操作系统：Linux、Freebsd、Darwin、Windows、Android、Dragonfly、Netbsd、Openbsd、Plan 9 和 Solaris。$GOARCH 指的是目标处理器的架构，目前支持的架构有 9 种：ARM、ARM64、386、AMD64、PPC64、PPC64le、MIPS64、MIPS64le 和

S390X，在构建时指定这两者就可以指定构建相应的 Go 语言程序。

如果要生成不同平台架构的可执行程序，只要改变这两个环境变量就可以了，比如要生成 Windows 下的 64 位 Go 语言程序，命令如下：

```
# 这里更改的环境变量只对当前 shell 会话有效，不会更改系统的配置。退出当前 shell 会话就会恢复
# 系统原来的配置
GOOS=windows GOARCH=amd64 go build code.local/hello-world
```

1.4 Go 语言的开发工具

虽然 Go 语言出现已经快有 10 年历史了，但是相比主流的 C/C++、Java、Python、PHP 语言，还是显得很年轻。成熟的 Go 语言集成开发环境并不多，本节将介绍几个著名的可用于 Go 语言开发的集成环境。

1.4.1 Code/Atom

1. Visual Studio Code

来自微软的开源编辑器 Visual Studio Code（以下简称为 Code）的确是开发 Go 语言程序的好工具之一，Code 的庞大生态使得它不仅仅是一个编辑器，它还拥有大部分 Go 语言开发过程中需要使用的工具链。

安装 Code 的 Go 语言插件可以在插件标签页中搜索，如图 1-2 所示。

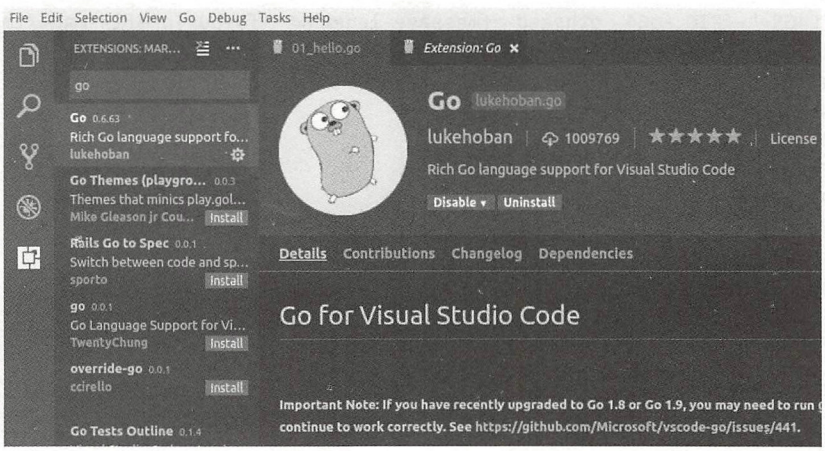


图 1-2　Code 安装 Go 语言插件

仅供非商业用途或交流学习使用

单击 Install 按钮即可安装，然后打开上面那个 Hello World 项目的源文件，此时 Code 检测到打开的是 Go 语言源代码，便会检查 Go 语言开发工具链是否完整，如果是新手一般都会不完整，此时 Code 会提醒你安装其他 Go 语言插件，这些插件实际上是通过 go get 安装的，这个时候前面提到的$GOPATH 就起作用了，Code 会把这些工具下载到$GOPATH 的第一个值的路径中。

因为国内网络的问题，可能需要为 Code 设置代理，或者在终端手动挂代理通过 go get 拉取相应工具。

```
$ go get -u -v github.com/nsf/gocode
$ go get -u -v github.com/rogpeppe/godef
$ go get -u -v github.com/zmb3/gogetdoc
$ go get -u -v github.com/golang/lint/golint
$ go get -u -v github.com/lukehoban/go-outline
$ go get -u -v sourcegraph.com/sqs/goreturns
$ go get -u -v golang.org/x/tools/cmd/gorename
$ go get -u -v github.com/tpng/gopkgs
$ go get -u -v github.com/newhook/go-symbols
$ go get -u -v golang.org/x/tools/cmd/guru
$ go get -u -v github.com/cweill/gotests
```

下面是如何为 Code 设置代理的方法，如图 1-3 所示。

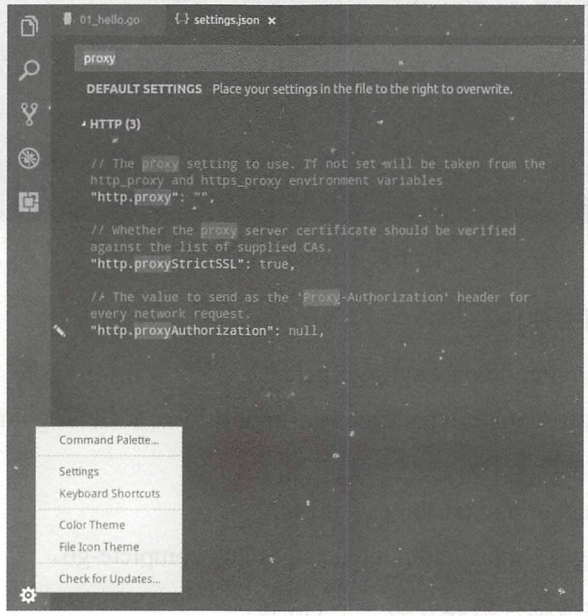


图 1-3　Code 代理

仅供非商业用途或交流学习使用

首先在 Code 界面左下角单击设置图标，打开设置界面，然后搜索 proxy 关键词，找到与代理相关的设置项，单击设置项左边的“编辑”按钮，会在右边的编辑框中出现一条用户设置模板（截图中未显示），在设置模板中填写你的代理设置信息即可。

2. Atom

与 Visual Studio Code 一样，Atom 同样是出色的编辑器，凭借其高人气的生态环境，要打造一个适合 Go 语言开发的环境自然不是什么难事。在 Atom 界面搜索 go，找到相关的插件安装。例如 go-config（https://atom.io/packages/go-config），这个是让 Atom 能使用 golang 的关键，安装后在这个插件的 Settings→GOPATH 中填写前面的$GOPATH 的路径，如图 1-4 所示。

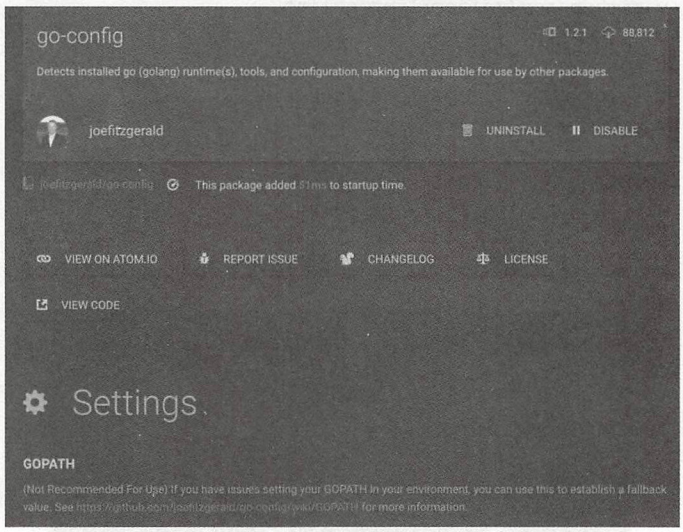


图 1-4　Atom 设置 GOPATH

接下来是其他辅助插件。

- go-get：https://atom.io/packages/go-get。
- go-import：https://atom.io/packages/go-imports（需要填写 goimports 的路径，它在 $GOROOT/bin 目录中）。
- gotests：https://atom.io/packages/gotests。
- autocomplete-go：https://atom.io/packages/autocomplete-go。

最后一个自动完成插件需要安装一个 Go 包：

```
$ go get -u github.com/nsf/gocode
```

仅供非商业用途或交流学习使用

现在 Atom 可以通过自动补全来编写 Go 语言程序了。最后也有更方便的安装方式，例如使用集成几个插件的插件 go-plus，直接安装这个插件把上面常用的 Go 语言开发工具链安装好。

1.4.2　VIM-Go

VIM 编辑器是大部分程序员的最爱之一，在 VIM 的生态中也有一款比较流行的 Go 语言开发插件——VIM-Go，安装这个插件与安装其他 VIM 插件的方法一样。下面是安装 Go 语言插件的演示。

根据不同 VIM 的插件管理工具，安装方式略有不同：

```
# Pathogen
$ git clone https://github.com/fatih/vim-go.git ~/.vim/bundle/vim-go
# vim-plug
Plug 'fatih/vim-go'
# Vim packages
$ git clone https://github.com/fatih/vim-go.git ~/.vim/pack/plugins/start/vim-go
```

安装完毕后，还需要安装所有必需的二进制文件。有一个方便的命令，打开 VIM，输入：

```
:GoInstallBinaries
```

现在编辑一个 Go 语言源代码，就会有提示了，如图 1-5 所示。

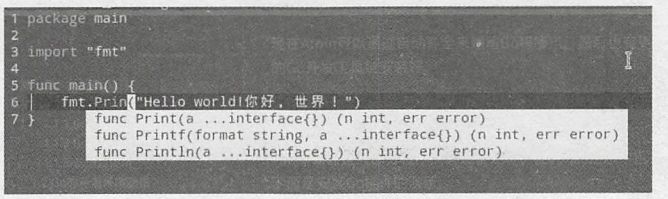


图 1-5　VIM 的 Go 语言扩展

启用这个插件之后，编辑源代码后保存，如果代码有错，是不能保存的，插件会提示哪里有错误。

如果希望一键配置整个 VIM 用于开发 Go 语言甚至其他语言，可以安装 EverVim：https://github.com/LER0ever/EverVim。

这个项目可以自动把默认的 VIM 配置为一个功能强大的小型 IDE。

仅供非商业用途或交流学习使用

1.4.3 Gogland

来自著名的 Jetbrains 公司开发的 IDE——Gogland 是一款专门为 Go 语言开发而设计的 IDE，作为 Jetbrains 的一员，Gogland 拥有 Jetbrains 家族的传统特色，如图 1-6 所示。

目前 Gogland 处于开发阶段，尚未正式发布，但是我们依旧可以在其官网免费下载最新的测试版本。Gogland 集成了编码助手（代码检查、自动补全、快速导航、格式化等）与一系列集成工具，最重要的是还支持一系列 IntelliJ 插件。

打开 https://www.jetbrains.com/go/download 页面，单击 DOWNLOAD 超链接即可下载，下载后解压，并把文件夹移到你喜欢的位置，最后执行 bin 目录里面的 gogland.sh 脚本即可启动。

因为 Gogland 是 IntelliJ 家族的一员，所以电脑必须安装了 Java 并配置了 JDK 环境（至少也要 JRE 环境），同时还需要手动配置$GOPATH，目前该软件暂时没有汉化版本。

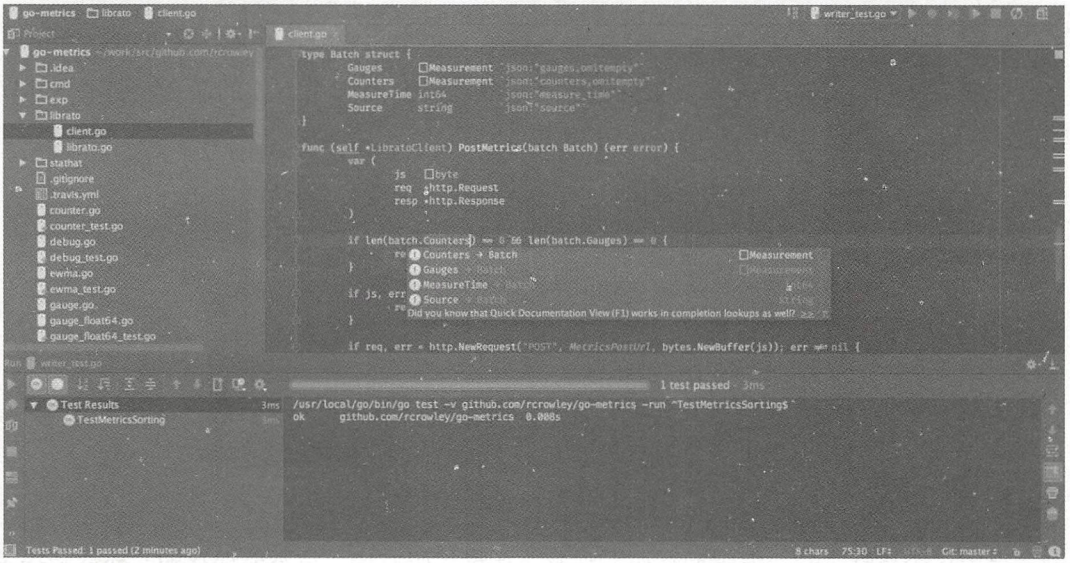


图 1-6 Gogland 界面

1.4.4 LiteIDE

LiteIDE（如图 1-7 所示）是一款专门为 Go 语言开发的跨平台轻量级集成开发环境（IDE），LiteIDE 是开源的工具，代码托管在 GitHub 上：https://github.com/visualfc/liteide。

仅供非商业用途或交流学习使用

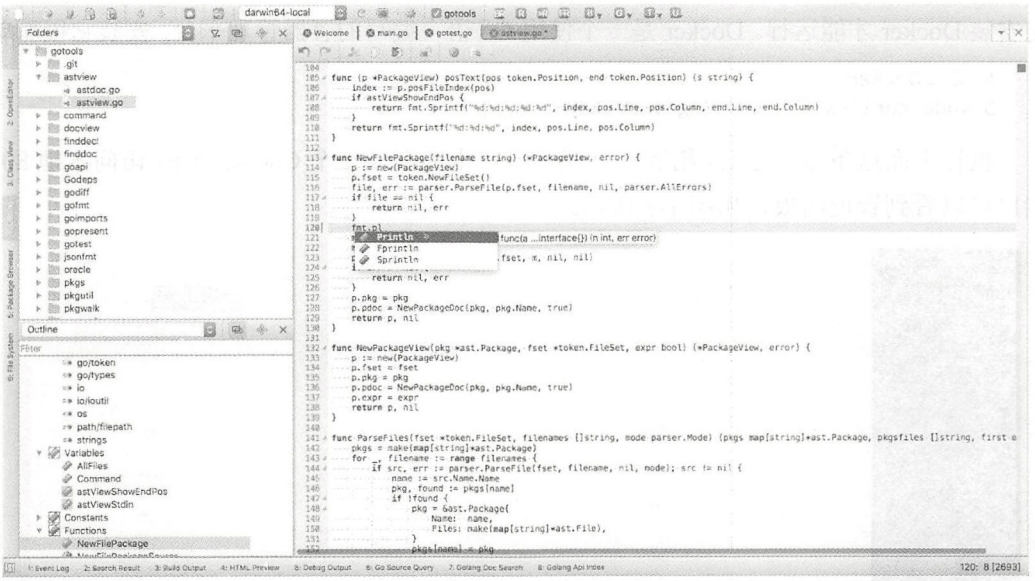


图 1-7　LiteIDE 界面

LiteIDE 支持主流操作系统（Windows、Linux、Mac OS），支持 Go 语言编译环境管理和切换，能够管理和切换多个 Go 语言编译环境，界面支持 Go 语言交叉编译。LiteIDE 提供了一个基于$GOPATH 的包浏览器和一个基于$GOPATH 的编译系统，能够通过 API 文档检索相应信息。

在编辑 Go 语言程序方面，拥有类浏览器和大纲显示功能，完美支持 Gocode（代码自动完成工具）与 Gdb 断点和调试，自动格式化代码。

下载地址：https://github.com/visualfc/liteide/releases。

首先安装好 Go 语言环境，然后根据操作系统下载 LiteIDE 对应的压缩文件直接解压即可使用。

1.4.5　Cloud IDE

1. Eclipse Che

号称下一代 Eclipse 的 Che 一直拥有较高的人气，Che 是一个 Web IDE，它支持主流的绝大部分编程语言，并且拥有一个完善的管理平台，简单来说就是一个私人用 PaaS 平台与 IDE 的结合。

快速启动 Che 只需一个命令：docker run eclipse/che start，这是一句 Docker 命令，需要

仅供非商业用途或交流学习使用

先安装 Docker 才能运行。Docker 是一个使用 Go 语言编写的容器管理平台，安装它很简单：

```
# 安装 Docker
$ sudo curl -sSL https://get.docker.com/ | sh
```

执行上面这条命令之后，稍等片刻就可以在本地启动一个 Che 实例了，访问本地 8080 端口可以看到管理面板，如图 1-8 所示。

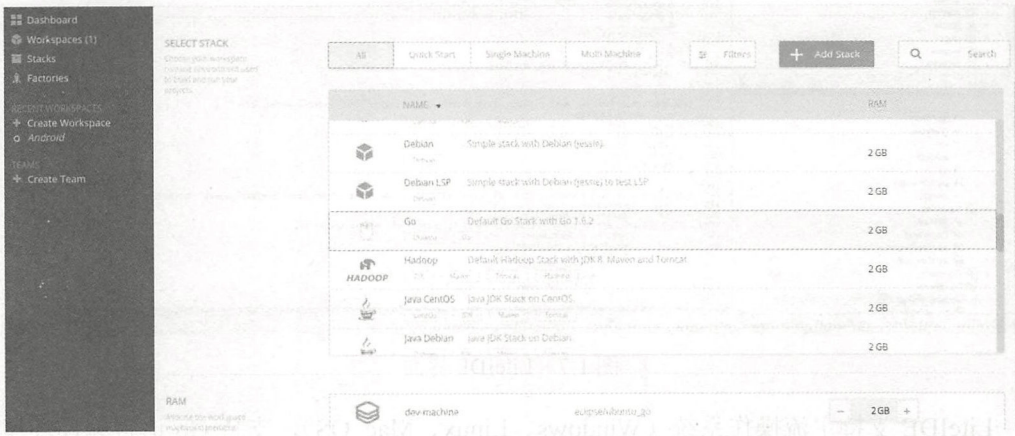


图 1-8　Eclipse Che 管理面板

新建一个 Go 语言工作区（如图 1-9 所示）之后导入两个示例项目（或者新建）。

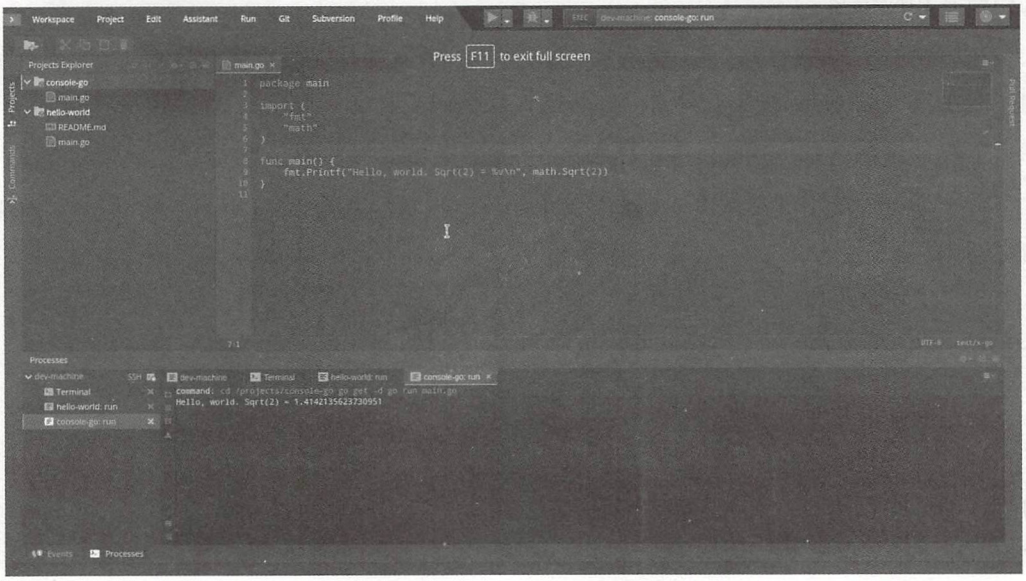


图 1-9　Eclipse Che 开发界面

仅供非商业用途或交流学习使用

如图 1-9 所示，Che 拥有出色的界面与丰富的功能。不过因为 Che 是高度依赖 Docker 的，所以需要掌握一些 Docker 的基础知识，才能更好地使用 Che。

2. Cloud9 IDE

Cloud9 IDE 是一个使用 Node.js 开发的 Web IDE，它支持主流的编程语言，包括 Go 语言，Cloud9 IDE 拥有出色的界面与强大的功能，集成了持续构建、即时预览、团队协作等功能。在此介绍如何在 Linux 下快速安装使用 Cloud9 IDE（以下内容基于 Ubuntu）。

```
# 安装 Docker
$ sudo curl -sSL https://get.docker.com/ | sh
# 启动 Cloud9 IDE
$ docker run -d -it --name=cloud9 \
      -v /workspace:/workspace \
      -p 8181:8181 zuolan/cloud9-ide --auth user:password
```

上面的命令，第一句表示安装 Docker，我们使用 Docker 运行一个 Cloud9 IDE 容器。第二句表示启动一个容器，这个容器名称为 cloud9，而且数据保存在这台计算机的 /workspace 目录，访问的端口号是 8181，用户名和密码分别是 user 和 password。

上面提到的几个关键值都可以修改，务必修改你的用户名和密码。根据网络情况，第一次打开 Cloud9 IDE 时可能会花费较长的时间，耐心等待。进入 IDE 界面后会看到一个欢迎界面，如图 1-10 所示，在这里可以做一些简单设置。



图 1-10　Cloud9 欢迎界面

仅供非商业用途或交流学习使用

可以单击右上角的设置按钮配置 IDE，Cloud9 IDE 功能很强大，不逊于 Eclipse Che，而且因为 Cloud9 IDE 是基于 Node.js 开发的，内存占用要小得多。浏览一遍设置，按自己习惯配置完成后，可以开始安装开发环境。

如果想自己编译一个 Cloud9 IDE 容器，可以参考下面的 Dockerfile：

```
FROM node:slim
MAINTAINER izuolan <i@zuolan.me>

RUN buildDeps='make build-essential g++ gcc python2.7' && softDeps="git" && \
    apt-get update && apt-get upgrade -y && \
    apt-get install -y $buildDeps $softDeps --no-install-recommends && \
    yarn global add forever && \
    git clone https://github.com/c9/core.git /cloud9 && cd /cloud9 && \
    scripts/install-sdk.sh && \
    apt-get purge -y --auto-remove $buildDeps && \
    apt-get autoremove -y && apt-get autoclean -y && apt-get clean -y && \
    rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*

VOLUME /workspace
EXPOSE 8181
ENTRYPOINT ["forever", "/cloud9/server.js", "-w", "/workspace", "-l", "0.0.0.0"]
#CMD["--auth","username:password"]
```

按 F6 键打开 IDE 终端，根据 Linux 安装 Go 语言环境的方法安装 Go 语言开发环境，如图 1-11 所示。

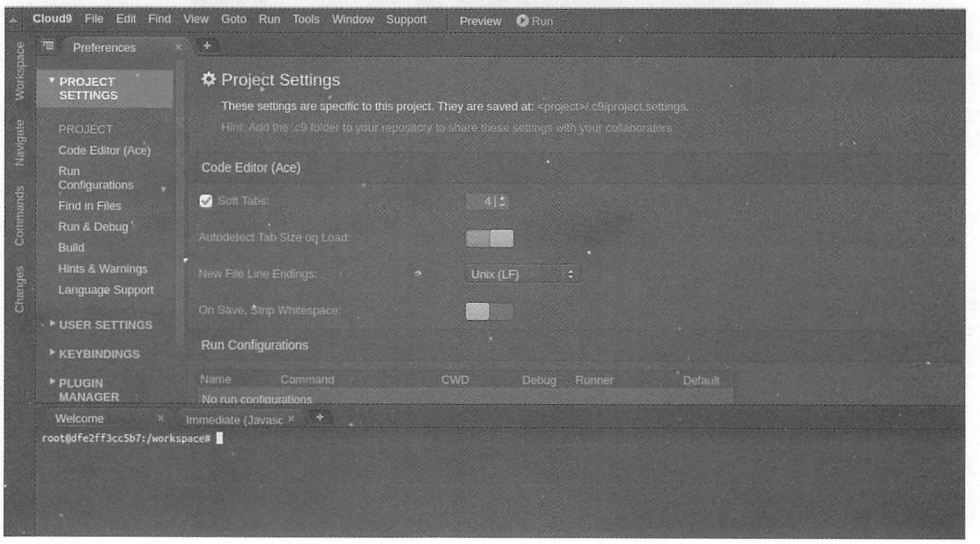


图 1-11　Cloud9 IDE 开发界面

仅供非商业用途或交流学习使用

3. Wide

Wide 是一个开源的基于 Web 的 Go 语言 IDE，由国人开发，可以使用官网搭建的服务，也可以下载到本地运行，如图 1-12 所示。

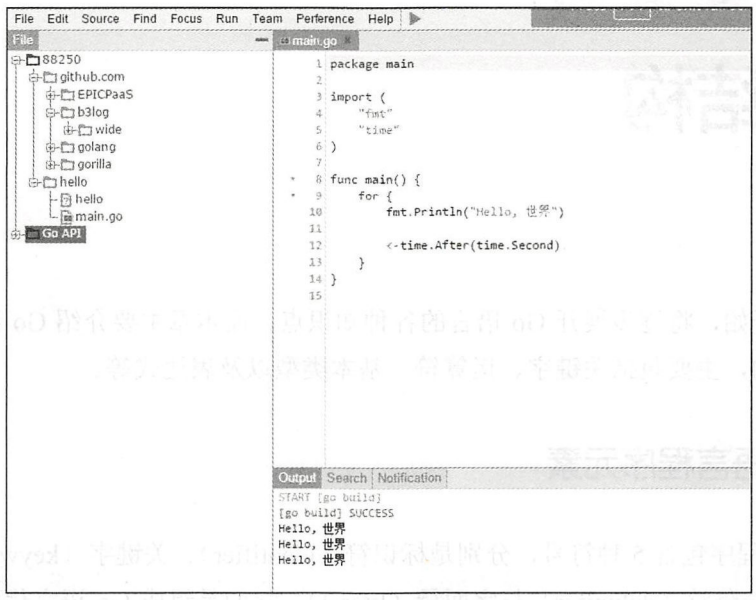


图 1-12　Wide 界面

对于开发工具，我们只选合适的不选最好的。除了本节介绍的几个开发工具，在 Go 语言官方仓库中，也总结了一些适合开发 Go 语言程序的工具，参见如下地址：

https://github.com/ golang/go/wiki/IDEsAndTextEditorPlugins

1.5　本章小结

本章介绍了 Go 语言的基本知识与安装方法，还推荐了几款适合开发 Go 语言程序的 IDE 或者编辑器。下一章会介绍 Go 语言的基本语法与程序结构。

仅供非商业用途或交流学习使用

第2章 程序结构

从本章开始，将逐步展开 Go 语言的各种知识点，而本章主要介绍 Go 语言的基本概念与程序结构，主要包括关键字、运算符、基本类型以及表达式等。

2.1 Go 语言程序元素

Go 语言程序包含 5 种符号，分别是标识符（identifier）、关键字（keyword）、运算符（operator）、分隔符（delimiter）与字面量（literal），它们是组成 Go 语言代码和程序的最基本单位，这 5 种符号共同组成各种表达式和语句。

2.1.1 标识符

Go 语言标识符是用于标识变量、函数或任何用户自定义项目的名称（程序实体）。标识符有两种类别，一种是用户定义的标识符，它作为程序实体存在，一般来说同一个代码块中不允许程序实体同名，使用不同代码包中的实体需要使用包标识符声明，例如前面遇到过的 fmt.Println 表示的是使用 fmt 包中的 Println 标识符。

另一种标识符称为预定义标识符，它们在 Go 语言源代码中声明。例如数据类型：bool（true 和 false）、byte、uint16、float32、float64、int、int8、int16、uint32、int32、int64、uint64、string、uint、uint8、uintptr；内建函数名：append、cap、close、complex、copy、delete、imag、len、make、new、panic、print、println、real、recover；其他标识符：iota、nil、_。

最后一个_叫作空标识符，一般用在变量声明与包的导入中，例如 Go 语言程序中，若存在一个没有使用的变量 a，那么编译时会报错，因为 Go 语言要求存在实体必须使用，

仅供非商业用途或交流学习使用

所以可以加一句_=a 这样的代码，这样 Go 语言编译器会认为使用了 a 变量，也就不会报错了，而实际上并没有对变量 a 做任何操作。程序初始化后空标识符就会被释放。

标识符以字母 a～z（大小写均可）或下画线_开头，后面可以是多个字母、下画线和数字。Go 语言不允许在标识符中使用标点符号，例如@、$、%等一系列符号。

虽然上面说大小写字母均可，但 Go 语言是一种区分大小写的编程语言，所以 Test 和 test 在 Go 语言程序中表示的是两个不同的标识符。

2.1.2　关键字

Go 语言作为一门静态编译型语言，它的关键字一共有 25 个，相比其他编程语言而言显得非常少，因此某种意义上也就更简单了。下面是 Go 语言的 25 个关键字。

- 包导入与声明：import、package。
- 程序实体声明与定义：var、type、func、interface、map、struct、chan、const。
- 流程控制：if、continue、for、return、go、case、goto、switch、select、else、break、default、defer、fallthrough、range。

上面显示的 Go 语言关键字，也称为保留字，这些保留字不能用作常量、变量或任何其他标识符名称。除此之外，还有一个比较特殊的类型叫作空接口，它的字面量是“_”（下画线）。

2.1.3　字面量

字面量就是表示值的一种标记法，但是在 Go 语言中，字面量的含义要更广一些。首先，它用于表示基础数据类型值的各种字面量；其次用于表示用户构造的自定义复合数据类型的类型字面量；最后它还用于表示复合数据类型的值的复合字面量，简单来说，它会被用来构造类型 Struct（结构体）、Array（数组）、Slice（切片）和 Map（字典）的值。

举一个通俗的例子，以 Go 语言为例：

```
const b int = 10 // b 为常量，10 为字面量
var str string = "hello world" // str 为变量，hello world 为字面量
```

2.1.4　分隔符

分隔符，正如其名，用于分隔其他元素，例如 fmt.Println("Hello, World!")中包含了 6 个标记：

```
fmt
.
Println
(
"Hello, World!"
)
```

程序中使用的括号分隔符包括括号、中括号和大括号。程序中使用的标点分隔符包括小数点、逗号、分号、冒号和省略号。

程序代码都是通过语句来实现结构化的，但是 Go 语言不需要以分号;结尾，这些工作由 Go 语言编译器自动完成。当然，如果将多个语句写在同一行，它们就必须使用;分隔，实际开发中不鼓励这种写法。

2.1.5 运算符

运算符是用于执行运算或逻辑操作的符号。Go 语言的运算符如表 2-1 所示。

表 2-1 Go 语言运算符说明

运算符	说　明
||	或，二元逻辑运算符，例如 true || false 返回的结果就是 true
&&	与，二元逻辑运算符，例如 true && false 返回的结果就是 false
!	非，一元逻辑运算符，例如!a 返回 true（a 的值为 false）
==	相等判断，二元逻辑运算符，例如"string" == "string"返回的结果是 true
!=	不等判断，二元逻辑运算符，例如"string" != "String"返回的结果是 true
<	小于判断，二元逻辑运算符，例如 1<2 返回的结果是 true
<=	小于等于判断，二元逻辑运算符，例如 2<=2 返回的结果是 true
>	大于判断，二元逻辑运算符，例如 2>1 返回的结果是 true
>=	大于等于判断，二元逻辑运算符，例如 2>=2 返回的结果是 true
+	求和操作，二元算术运算符，例如 1+1 返回 2
-	求差操作，二元算术运算符，例如 1-1 返回 0
*	乘积操作，二元算术运算符，例如 2*3 返回 6。同时也是一元地址运算符，表示取值操作
/	求商操作，二元算术运算符，例如 6/3 返回 2
|	按位或操作，二元算术运算符，例如 5 | 11 返回 15
^	按位异或操作，二元算术运算符，例如 5 ^ 11 返回 14。同时也是一元运算运算符，表示按位取补码操作，例如^5 返回-6
%	求余操作，二元算术运算符，例如 12%5 返回 2
<<	按位左移操作，二元算术运算符，例如 4<<2 返回 16

续表

运算符	说　明
>>	按位右移操作，二元算术运算符，例如 4>>2 返回 1
&	按位与操作，二元算术运算符，例如 5&11 返回 1。同时也是一元地址运算符，表示取址操作，例如 &str 返回 str 所在的内存地址
&^	按位清除操作，二元算术运算符，例如 5&^11 返回 4

当一个表达式中有多个运算符时，就会遇到优先级的问题，在 Go 语言中，一元运算符的优先级高于二元运算符，二元运算符的优先级如表 2-2 所示。

表 2-2　二元运算符优先级

运 算 符	优 先 级
*　/　%　<<　>>　&　&^	最高
+　-　|　^	较高
==　!=　<　<=　>　>=	中
&&	较低
||	最低

如果一个表达式中出现相同优先级的运算符，那么根据从左到右的顺序依次操作。当然，所有运算符都受括号的影响，括号内先操作，这和平常的数学运算法则无异。需要注意的是，在 Go 语言中，++和--是语句，不是表达式，没有运算符优先级之说。

```
func main() {
	n := 0
	p := &n
	// b := n++
	// if n++ == 1 {}
	// ++n
	n++
	*p++
	fmt.Println(n)
}
// 上面注释的都是语法错误的，:=是一种简写，表示函数内部变量声明并赋值
```

最后，Go 语言不支持运算符重载。

2.1.6　注释

Go 语言编译器遇到注释会完全忽略。注释是源代码中最重要的组成部分之一，注释的用途非常广泛，例如可以填写供人类阅读的短小精悍的解释文档，或者可以暂时让编译器忽略部分源代码。

Go 语言的注释有多种写法：

```
fruit = apples + oranges;   // 获取所有水果数量
// 获取所有动物数量
animals = dogs + cats;
/* 这是一条 Go 语言注释 */
/* 这是
一段 Go 语言
注释 */
```

以上注释的写法均可。

2.2 Go 语言基本概念

在学习 Go 语言之前，需要了解 Go 语言的一些基本概念，包括如何定义常量、变量，程序实体的命名有怎样的规则，以及声明一个实体有哪些需要注意的地方。

2.2.1 常量

在 Go 语言中，常量是指编译期间就已知，且在执行过程中不会改变的固定值。在 Go 语言程序中，常量可定义为数值（整型、浮点型和复数类型）、布尔值或字符串等类型。常量使用关键字 const 定义，常量的定义格式：const identifier [type] = value，例如：

```
const Pi = 3.1415926536
// 也可以明确指定常量的类型
const Pi float32 = 3.1415926536
// 也可以是布尔值、字符串等
const Return = true
const hello = "你好，Go 语言。"
// 甚至可以使用中文
const 人数 = 100
```

在上面例子中，你会发现可以省略类型说明符[type]，Go 语言的编译器可以智能地判断值的类型。

- 显式定义：const h string = "hello"
- 隐式定义：const w = "world"

除此之外，Go 语言常量比较有趣的一点就是支持超长的小数位数，但是若指定了常量为 float32 类型，Go 语言编译器就会自动缩短常量为 32b，若是指定了常量为 float64 类型，则 Go 语言编译器会自动缩短为 64b。而且 Go 语言支持反斜杠\，可以在定义常量的

表达式中使用反斜杠作为跨行的连接符使用，这与 Linux 的 shell 命令一样。

```
const Pi = 3.141592653589793\
           23846264338327950288\
           419716939937510582097494 4\
           592307816406286208998628034825342117067 9
```

前面说了常量的值必须是固定的，这个固定是指在编译时可以确定，也就是说可以在其赋值表达式中涉及计算过程，但是必须在编译期间就能确定所有用于计算常量的值。

```
const c1 = 6/3  // 返回 2
// const c2 = getNumber(member)  // getNumber 是用户自定义函数，不确定
/* 在编译期间，自定义函数返回的值未知，所以不能赋值给常量。
   但内置函数可以，例如：len()函数。 */
n := "hello"
l := len(n) // 返回 5
```

有意思的是，与 Go 语言的数字变量相比，Go 语言的数字常量不需要考虑类型转换问题，但需要注意常量赋值给精度过小的数字变量而导致溢出的情况，这会在编译期间报错。

1. 数字表示

Go 语言的常量可以是十进制、八进制或十六进制常数。

- 前缀是 0x 或 0X 为十六进制。
- 前缀是 0 的为八进制。
- 十进制无前缀。

整数还可以有一个后缀，它是 U 和 L 的组合，分别用于表示 unsigned 和 long。后缀可以是大小写字母，顺序也是任意的。示例如下：

```
3.14        // 十进制，合法
0213        // 八进制，合法
0x4b        // 十六进制，合法
0xFeeL      // 合法
30u         // 无符号整型，合法
30l         // long，合法
30ul        // 无符号 long，合法
078         // 非法：8 is not an octal digit
032UU       // 非法：cannot repeat a suffix
```

2. 转义字符

与其他语言一样，Go 语言使用反斜杠表示转义符号，例如换行符（\n）或制表符（\t）。表 2-3 所示为常见的转义代码列表。

表 2-3　常见转义代码列表

转义序列	含　义
\\	\字符
\'	'字符
\"	"字符
\?	?字符
\b	退格
\f	换页
\n	新行
\r	回车
\t	水平制表符
\v	垂直制表符

示例：

```
package main

import "fmt"

func main() {
      fmt.Println("Hello\tWorld!")
      fmt.Println("Hello\rWorld!")
}
```

当上述代码被编译和执行时，它产生以下结果：

```
Hello World!
Hello
World!
```

3. 赋值

Go 语言支持以下赋值运算符：

```
package main

import "fmt"

func main() {
      a := 20
      c := 200
      c = a
      fmt.Println("赋值操作，把 a 赋值给 c，所以 c 的值为：", c)
      c += a
      fmt.Println("相加和赋值运算符，实际为 c = c + a，所以 c 的值为：", c)
      c -= a
```

```
    fmt.Println("相减和赋值运算符，实际为 c = c - a，所以 c 的值为：", c)
    c *= a
    fmt.Println("相乘和赋值运算符，实际为 c = c * a，所以 c 的值为：", c)
    c /= a
    fmt.Println("相除和赋值运算符，实际为 c = c / a，所以 c 的值为：", c)
    c <<= 2
    fmt.Println("左移和赋值运算符，所以 c 的值为：", c)
    c >>= 2
    fmt.Println("右移和赋值运算符，所以 c 的值为：", c)
    c &= 2
    fmt.Println("按位与和赋值运算符，所以 c 的值为：", c)
    c ^= 2
    fmt.Println("按位异或和赋值运算符，所以 c 的值为：", c)
    c |= 2
    fmt.Println("按位或和赋值运算符，所以 c 的值为：", c)
}
```

返回结果是：

```
赋值操作，把 a 赋值给 c，所以 c 的值为： 20
相加和赋值运算符，实际为 c = c + a，所以 c 的值为： 40
相减和赋值运算符，实际为 c = c - a，所以 c 的值为： 20
相乘和赋值运算符，实际为 c = c * a，所以 c 的值为： 400
相除和赋值运算符，实际为 c = c / a，所以 c 的值为： 20
左移和赋值运算符，所以 c 的值为：%!(EXTRA int=80)

右移和赋值运算符，所以 c 的值为： 20
按位与和赋值运算符，所以 c 的值为： 0
按位异或和赋值运算符，所以 c 的值为： 2
按位或和赋值运算符，所以 c 的值为： 2
```

与其他现代编程语言一样，Go 语言支持并行赋值的写法：

```
const 人数，费用，班级 = 40，200.01，"一班"
const (
    Monday，Tuesday，Wednesday = 1，2，3
    Thursday，Friday，Saturday = 4，5，6
)
```

4. 枚举

常量还可以用作枚举：

```
const (
    Connected = 0
    Disconnected = 1
    Unknown = 2
)
```

现在，数字 0、1、2 分别代表连接成功、连接断开和未知。在下面例子中，使用 iota

关键字实现枚举：

```
package main

import (
      "fmt"
)

const (
      a       = iota              // a == 0
      b                           // b == 1，隐式使用 iota 关键字，实际等同于 b = iota
      c                           // c == 2，实际等同于 c = iota
      d, e, f = iota, iota, iota  // d=3,e=3,f=3，同一行值相同，此处不能只写一个 iota
      g       = iota              // g == 4
      h       = "h"               // h == "h"，单独赋值，iota 依旧递增为 5
      i                           // i == "h"，默认使用上面的赋值，iota 依旧递增为 6
      j       = iota              // j == 7
)

const z = iota // 每个单独定义的 const 常量中，iota 都会重置，此时 z == 0

func main() {
      fmt.Println(a, b, c, d, e, f, g, h, i, j, z)
}
```

每个 const 定义的第一个常量被默认设置为 0，显式设置为其他值除外。后续的常量默认设置为它上面那个常量的值，如果前面那个常量的值是 iota，则它也被设置为 iota。因为 iota 有递增效果，所以可以实现枚举。此外，iota 也可以用在表达式中，如：iota + 50。

2.2.2 变量

1. 变量的声明

变量是 Go 语言中表示抽象值和储存计算结果的一种概念。变量的值通过变量名访问，Go 语言的变量名命名规则与常量命名一样，都是由字母、数字、下画线组成的，其中名称首个字符不能为数字等。

变量的声明格式是：var identifier type，也就是使用 var 关键字。从前面定义常量的小节中，就已经见识过了 Go 语言和其他大部分编程语言不同的一个地方，就是它在声明变量时将变量的类型放在变量的名称后面。这样的定义语法有多种原因，其中一点是为了避免像 C 语言中出现有歧义的声明情况，具体讨论详见：http://blog.golang.org/2010/07/gos-declaration-syntax.html。

另外，这种语法从左至右的顺序阅读，更符合人类语言习惯，例如：

```
// 定义一个变量 a 为 int 型
var a int
// 定义一个变量 b 为 bool 型
var b bool
// 定义一个变量 str 为 string 型
var str string
// 定义两个变量 a、b 为 int 型
var a, b int
```

也可以与常量一样写成这种形式：

```
var (
      a, b int
      c    bool
      str  string
)
// 同一类型的多个变量可以声明在同一行
var a, b, c int
// 多变量可以在同一行进行声明和赋值
var a, b, c int = 1, 2, 3
// 多变量可以在同一行进行赋值（只能在函数体内）
a, b = 5, 7
```

如果想要交换两个变量的值，可以使用 a, b = b, a 语句。

上面的声明方式也叫静态声明，除此之外还有动态声明方式，与声明常量一样，例如：

```
var a, b, c = "string", 1, true
```

在这种定义声明中，三个变量都拥有不同的类型，Go 语言编译器会自动推断。

2. 局部变量声明

变量的声明有两种写法，上面是一种，另一种形式是使用:=赋值运算符号代替，例如：

```
a := 1
// 等同于 var a = 1
```

前面说过 Go 语言编译器可以智能判断类型，此处不写 a 变量的类型也是因为 Go 语言编译器会自动判断类型。

Go 语言官方推荐使用这种方式定义变量，但是必须注意，它只能被用在函数体内，不可以用于全局变量的声明与赋值（注释 1、2 和 4）。

```
package main

import (
```

```
    "fmt"
)

/* 1. 这种全局定义的写法是错误的。 */
// a := 1

/* 2. 全局定义使用这种方式。 */
var a = 1

/* 3. 前面已经定义了变量 a，同一个代码块中不能再次定义。 */
// a := 10

func main() {
    b := 2  // 4. 只能在函数体内部使用简写形式
    a = 100 // 5. 把 10 赋值给 a 是没有语法问题的
    fmt.Println(a)

    a := "hello" // 6. 函数体内部属于另一个代码块，可以再次声明变量 a，改变变量类型

    fmt.Println(a, b, c)

    /* 7. 不能在调用之后定义变量 */
    // fmt.Println(d)
    // d := 4
}

// 8. 全局定义不限定位置，在函数之外即可
var c = 0

/* 返回:
100   2   0
*/
```

使用:=可以快速创建一个函数内部的变量，它包括初始化声明与赋值两大部分。注意以下几点：

- 同一个代码块内不能多次声明同一个变量（注释 3）。
- 函数内部可以使用全局定义的变量（注释 5），意思是给变量赋予一个新的值。
- 函数体内属于独立代码块，可以再次声明变量 a（注释 6），可以改变变量的类型，并且定义只在函数内部生效。
- 如果在定义变量之前就调用该变量，那么会编译错误，返回 undefined: a（注释 7）。
- 全局定义不限制位置，但建议统一放在函数代码前面，见注释 8。

当变量被声明但没有赋值时，它的值就是定义时指定类型的零值，例如 int 为 0、float

为 0.0、bool 为 false、string 为空字符串、指针为 nil 等。如果变量甚至没有指定类型，那么系统默认它为 int 型，值为 0，所有变量在 Go 语言中都是需要初始化的。

变量的命名规范一般遵循驼峰命名法：首个单词小写，每个独立单词的首字母大写，例如：setNum 和 backTop（常量命名建议全部为大写字母，遵循 Shell 环境的习惯）。

当然命名规范并不是强制性规定，实际上在 Go 语言中，程序实体（常量、变量、函数）首字母大小写是有特殊含义的（外部可用性），这些在后面章节都会介绍（例如函数章节）。

变量的类型也可以在运行时实现自动推断，例如：

```
package main

import (
      "fmt"
      "os"
)

var (
    HOME = os.Getenv("HOME")
    USER = os.Getenv("USER")
    GOROOT = os.Getenv("GOROOT")
)

func main() {
      fmt.Println(HOME, USER, GOROOT)
}
```

上面这个例子展示了如何在运行时获取所在的操作系统类型，它通过 os 包中的函数 os.Getenv()来获取环境变量中的值，并保存到定义的全局变量中（读者可以在不同操作系统上执行这个程序，返回的结果会根据操作系统不同而不同）。

3. 匿名变量

全局变量自然是全局可用，局部变量只能在相应代码块内起作用，除此之外，局部变量还有一个特点就是，定义了必须使用（全局变量允许声明但不使用）。如果定义了局部变量，但是却没有在这个代码块中调用这个局部变量，那么编译也会出现错误，例如：

```
func main() {
   a  := 1000
   fmt.Println("hello, world")
}
```

尝试编译这段代码将得到错误 a declared and not used，这里的要求是使用，所以即使

是赋值也会出现编译错误，这个值必须被使用，例如打印出来，或者赋值给其他变量，或者加入某个运算中等都可以。

但在使用强类型语言编程时，经常会出现在调用函数时，该函数返回多个值，其中只有一个值是需要的情况。这个时候也有一个办法可以解决这种问题，那就是前面说过的空标识符，例如_=a，这样就不会报错了，这种写法可以忽略一些多返回值的函数的部分返回值。

匿名函数的写法可以让代码看起来更加优雅，例如下面的函数 GetClass()返回三个值，分别是 stuNum、className 和 headTeacher：

```
package main

import (
	"fmt"
)

// GetClass 获取班级信息
func GetClass() (stuNum int, className, headTeacher string) {
	return 49, "一班", "张三"
}

func main() {
	stuNum, _, _ := GetClass() // 只想获取班级人数 stuNum，其他不需要
	fmt.Println(stuNum)
}
```

这种用法可以让代码更加简洁，看起来更容易理解。

4. init 函数

Go 语言中，除了可以在全局声明中初始化实体，也可以在 init 函数中初始化。init 函数是一个特殊的函数，它会在包完成初始化后自动执行，执行优先级比 main 函数高，并且不能手动调用 init 函数。每一个源文件有且只有一个 init 函数，初始化过程会根据包的依赖关系按顺序单线程执行。

可以在开始执行程序之前通过这个函数来对数据进行检验与修复，保证程序执行时状态正常。例如：

```
package main

import (
	"fmt"
	"math"
```

```
)

// Pi 为圆周率
var Pi float64

func init() {
      Pi = 4 * math.Atan(1) // 在 init 函数中计算 Pi 的值
}

func main() {
      DPi := Pi * Pi
      fmt.Println(Pi, DPi)
}
```

以上程序返回：3.141592653589793 9.869604401089358。

2.3　本章小结

本章解释了 Go 语言的基本概念与程序结构，主要包括关键字、运算符以及表达式等，还介绍了 Go 语言中各种运算符的作用与用法。后面还对常量与变量的定义进行了详细的说明。本章遇到了很多不同的数据类型，以及写了不少函数，这些也属于 Go 语言程序的一部分，下一章将开始数据类型的学习。

第 3 章

基础数据类型

在 Go 编程语言中，数据类型用于声明函数与变量、常量的数据的类型。数据类型这个概念主要是为了提高内存的利用效率，因为不同数据存储在内存中所需要的空间大小是不一样的，所以编程时指定数据类型，可以充分利用内存空间，数据存储体积大的类型申请的内存就大一些，数据体积小一点的类型申请的内存就小一些。数据类型确定了程序实体在内存的存储中可以占用多少空间以及如何存储。

Go 语言内置基础类型如表 3-1 所示。

表 3-1　Go 语言基础类型列表

类型名称	类型和描述
boolean	布尔型，值只可以是常量 true 或者 false。例如：var b bool = true。类型长度为 1 字节
int/uint	整数类型（简称整型），Go 语言支持无符号和带符号两种整型，这两种整型的内存大小相等，但具体大小取决于编译器的实现。Go 语言也有已经定义了位数的整型：int8、int16、int32 (rune)、int64 和 uint8 (byte)、uint16、uint32、uint64。rune 是 int32 的别称，byte 是 uint8 的别称。8 位整型长度为 1 字节，16 位整型长度为 2 字节，32 位整型长度为 4 字节，64 位整型长度为 8 字节。uintptr 类型在指针小节中单独介绍
float32/float64	浮点型，浮点数的类型有 float32 和 float64 两种，注意 Go 语言没有 float 这种类型，默认是 float64，长度分别为 4 字节和 8 字节。此外还有两个复数类型 complex64、complex128，其中 complex64 类型带有 float32 实部和虚部，complex128 类型带有 float64 实部和虚部
string	字符串类型，字符串就是一串固定的字符序列。Go 语言的字符串是由单字节连接而成的，所有字节默认使用 UTF-8 编码标识 Unicode 文本。在 Go 语言中字符串是固定的，即不可变化
派生类型	派生类型，包括切片类型（slice）、字典（map）类型、通道类型（channel）、指针类型（pointer）、数组类型（array）、结构化类型（struct）、函数类型（function）、接口类型（interface）以及错误类型（error）。它们内部结构复杂，不仅需要申请内存，还需要初始化相关属性

Go 语言支持使用八进制、十进制、十六进制以及科学计数法表示数字。标准库 math 中定义了各数字类型的取值范围。例如 int8 的范围是-128～127，uint8 的取值范围是 0～255；int16 的取值范围是-32768～32767，uint16 的取值范围是 0～65535。

3.1　整型

Go 语言严格来说一共有 9 种不同大小的整型，无符号 4 种，有符号 4 种，还有一种 uintptr 类型（多用于底层编程）。其中 int8、int16、int32 和 int64 这 4 种不同大小的有符号整型数类型分别对应 8、16、32、64b 大小的有符号整型数；对应的无符号整型分别是 uint8、uint16、uint32 和 uint64。除了指定整型位数以外，也可以直接使用 int 和 uint 类型。

Go 语言可以根据不同平台的实现对 int 做出调整，既可以是 int32，也可以是 int64，uint 同理。

3.1.1　整型的表示

在 Go 语言中所有类型声明格式都是一样的：

```
var value1 int32 // 全局声明
func main() {
      value2 := 64 // 函数内部声明
}
```

第 2 章已经介绍了如何声明变量和常量，需要注意的是 int 和 int8 不是同一种类型，两种类型不能直接比较、不能直接运算，如果需要对两个不同类型的值执行运算操作，需要转换类型，这在稍后会介绍。

3.1.2　整型的运算

Go 语言的整型支持常规整数运算，加减乘除自不必多说，Go 语言与 C 语言一样，使用%表示求余符号，用作求余运算，比如：

```
5 % 3 // 结果：2
```

除了算术运算，常见的还有比较运算，　>、<、==、>=、<=和!=等比较符号 Go 语言都支持（没有三个等号的全等符号）：

```
x, y := 2, 4
if x == y {
```

```
        fmt.Println("x 等于 y")
    }
```

上面是一个 if 判断流程，当 x 等于 y 时输出“x 等于 y”，否则什么都不输出，尝试修改 x 和 y 的值查看输出。前面说过两个不同类型的整型数不能直接比较（见下面例子），但各种类型的整型变量都可以直接与字面常量（指源程序中表示固定值的符号，例如各种进制下的数字）进行比较，例如：

```
package main

import "fmt"

var x int32
var y int64

func main() {
        x, y = 2, 4
        if x == y { // 编译不通过
                fmt.Println("x 等于 y")
        }
        if x == 2 || y == 2 { // 编译通过
                fmt.Println("x 等于 y")
        }
}
```

编译不通过的 if 语句是因为尝试比较两个不同的类型，自然不能通过；编译通过的 if 语句中，字面常量两个 2 虽然看起来相等，但所表示的其实是不同的类型，在编译运行时，这两个字面常量都会被转换为相应的类型。可以把上面编译通过的代码改为如下形式验证：

```
        if x == 9999999999 || y == 2 { // 编译不通过
                fmt.Println("x 等于 y")
        }
```

编译时会得到“constant 9999999999 overflows int32”的错误，因为 9999999999 已经超过 int32 的范围。

Go 语言整型数据也支持位运算，如表 3-2 所示。

表 3-2　Go 语言整型数据所支持的位运算

运　算	例　子	结　果
x << y，左移	1 << 2	4
x >> y，右移	12 >> 2	3
x ^ y，异或	1 ^ 2	3

续表

运　算	例　子	结　果
x & y，与	1 & 2	0
x | y，或	12	2
^x，取反	^2	-3

使用方法与主流编程语言并无差异。

3.2　浮点型

浮点型用于表示包含小数点的数据，比如 3.14 就是一个浮点型数据。Go 语言提供了两种精度的浮点数：float32 和 float64，在 Go 语言中没有 float 这种类型。float32 精确到小数点后 7 位，float64 精确到小数点后 15 位。由于精度不同的缘故，在使用==或者!=来比较浮点数时需要特别注意，事先测试运算可以避免不少问题。

由于标准库 math 包中所有有关数学运算的函数都要求接收 float64 这个类型，所以尽可能使用 float64，省去后续类型转换的工作。

3.2.1　浮点型的表示

Go 语言定义一个浮点型变量：

```
package main

import (
	"fmt"
	"reflect"
)

func main() {
	var value1 float64
	value1 = 1    // 不管是否有小数点，value1 都是浮点型
	value2 := 2   // 如果没有小数点，value2 会被推导为整型
	value3 := 3.0 // 如果没有小数点，value3 会被推导为浮点型
	// v := value1 + value2 // 编译失败，value1 与 value2 类型不同
	v := value1 + value3
	fmt.Println(value1, value2, value3, v)
	fmt.Println("v 的类型是：", reflect.TypeOf(v))
}
/*
返回：
1 2 3 4
```

仅供非商业用途或交流学习使用


```
v 的类型是： float64
*/
```

上面使用了反射（reflect）的概念返回变量的类型，修改 TypeOf 中的变量名称，查看其他值的类型是否如你所想。

3.2.2 浮点型的运算

浮点数的加减乘除运算和整型没有差别，不同的是浮点型有些特殊，因为浮点型的精度不同会导致比较结果与预期不一样。例如：

```
package main

import (
	"fmt"
)

func main() {
	var value1 float64
	value1 = 1
	value2 := 1.0000000000000000001
	if value1 == value2 {
		fmt.Println("相等")
	}
}
/*
返回:
相等
*/
```

上面两个变量显然不相等，但是因为精度限制在小数点后 15 位，之后的数字已经被舍弃，所以程序会返回相等的结果。

3.2.3 复数

我们把形如 $a+bi$（a 和 b 均为实数，类型为浮点型）的数称为复数，其中 a 称为实部，b 称为虚部，i 称为虚数单位。Go 语言的复数类型关键字有两个：

```
complex64 // 32 位实数和虚数
complex128 // 64 位实数和虚数
```

复数表示的示例如下：

```
package main
```

仅供非商业用途或交流学习使用


```
import (
      "fmt"
)

var value1 float64

func main() {
      var v1 complex64 // 由两个 float32 构成的复数类型
      v1 = 3.2 + 12i

      v2 := 3.2 + 12i        // v2 隐式声明，默认是 complex128 类型
      v3 := complex(3.2, 12) // v3 数值等于 v2
      v := v2 + v3           // 复数运算
      fmt.Println(v1, v2, v3, v)
}

/*
返回：
(3.2+12i) (3.2+12i) (3.2+12i) (6.4+24i)
*/
```

上面 v1 实部和虚部均为 float32 类型，v2 和 v3 虽然不使用方法定义，但是返回的结果其实是相等的。使用函数 real(v)和 imag(v)可以分别获取相应的实部和虚部的数值：

```
package main

import (
      "fmt"
)

func main() {
      v2 := 3.2 + 12i        // v2 是 complex128 类型
      v3 := complex(3.2, 12) // v3 结果同 v2
      v := v2 + v3           // 复数运算
      fmt.Println(v2, v3, v)

      vr := real(v) // 获取实部
      vi := imag(v) // 获取虚部
      fmt.Println(vr, vi)
}
/*
返回：
(3.2+12i) (3.2+12i) (3.2+12i) (6.4+24i)
6.4 24
*/
```

更多关于复数的函数可查阅 math/cmplx 标准库的文档。复数支持常规数学运算，但因为实部和虚部是浮点型，在使用==、!=比较时，需要注意数字精度。cmath 包中包含了

仅供非商业用途或交流学习使用

一些操作复数的公共方法，除非是对内存有严格限制的程序，否则建议使用 complex128 作为声明类型，减少后续类型转换的操作。

3.3 字符与字符串

在 Go 语言中，字符串也是一种基本类型。这一点和 C 语言不同，C 语言没有原生的字符串类型，C 语言使用字符数组来表示字符串，并以字符指针来传递字符串。

在 Go 语言中，一个字符串是一个不可改变的 UTF-8 字符序列，一个 ASCII 码占用 1 字节，其他字符根据需要占用 2～4 字节。前面说过 Go 语言作者就是当年搞出 UTF-8 编码格式的那几个人，所以 Go 语言默认就是采用 UTF-8 编码格式进行文件存储的。由于这种编码会因不同字符而占用不同长度的字节，所以 Go 语言中的字符也可能根据需要占用 1～4 字节，这与其他一些主流语言（如 C++、Java、Python、PHP）不同。这样设计的好处有两个，首先可以减少内存的使用，节约硬盘空间，其次统一编码格式有助于减少读取时的编码和解码工作。

3.3.1 字符串的表示

在 Go 语言中，字符串的值是不可变的，当创建一个字符串之后，无法再次修改这个字符串的内容，在 Go 语言设计中，字符串是一个定长的字节数组，详见下面的例子。

Go 语言中字符串的声明和初始化非常简单：

```
package main

import (
	"fmt"
)

var value1 float64

func main() {
	s := "aA 你 2"
	fmt.Println("字符串长度：", len(s))
	for i := 0; i < len(s); i++ {
		fmt.Println(s[i])
	}
	/* 编译错误，字符串的值不能修改 */
	s := "你好，"
	t := s
```

仅供非商业用途或交流学习使用


```
    s += "世界。" // 字符串可以连接，但原字符串不会改变
    fmt.Println(s)
    // s[0] = 'L'
    fmt.Println(t) // 原字符串没有改变
}

/*
返回:
字符串长度： 6
97   // a
65   // A
228  // 你
189  // 你
160  // 你
50   // 2
你好，世界。
你好，
*/
```

上面代码中，定义了一个字符串 s，内置的 len()函数可以返回一个字符串中的字节数目（不是 rune 字符数目），索引操作 s[i]可以返回第 i 字节的字节值，i 必须满足 0≤i<len(s)条件约束。

Go 语言字符串类型支持以下两种形式的字面值。

1）解释字符串

该类字符串使用" "括起来，其中的相关的转义字符将被替换，这些转义字符包括：

```
\a      // 响铃
\b      // 退格
\f      // 换页
\n      // 换行
\r      // 回车
\t      // 制表符
\u      // Unicode 字符
\v      // 垂直制表符
\'      // 单引号          (只用在  '\''     形式的 rune 符号面值中)
\"      // 双引号          (只用在  "..."    形式的字符串面值中)
\\      // 反斜杠
```

2）非解释字符串

该类字符串使用反引号（一般是在 Esc 键下面，数字 1 键左边）括起来，支持换行。

```
package main

import (
    "fmt"
```

仅供非商业用途或交流学习使用


```
)

var value1 float64

func main() {
	str1 := `苟利国家生死以\n
	岂因祸福避趋之`
	str2 := "今天天气\n真好"
	fmt.Println(str1)
	fmt.Println(str2)
}

/*
返回:
苟利国家生死以\n
	岂因祸福避趋之
今天天气
真好
*/
```

注意上面反单引号和双引号的区别。首先反单引号可以跨行，并且引号内的所有内容都会直接输出，包括转义字符和空格缩进等。而双引号则不能换行，并且会解析转义字符。

3.3.2 操作字符串

前面说过字符串是不能修改的，但是可以通过索引逐个读取它的值：

```
package main

import (
	"fmt"
)

var value1 float64

func main() {
	s := "abcd你"
	for i := 0; i < len(s); i++ {
		fmt.Println(s[i])
	}

	c := s[len(s)]
	fmt.Println(len(s), c)

	fmt.Println(s[0:7])
	// fmt.Println(s[4:]) // 返回: 你
	// fmt.Println(s[:3]) // 返回: abc
```




仅供非商业用途或交流学习使用




```
}

/*
返回:
97
98
99
100
228
189
160
7 160
abcd 你
*/
```

从以上字符串可以看到，汉字占 3 字节，一共 7 字节长度，如果试图访问超出字符串索引范围的字节将会导致 panic 异常，例如把上面第 15、16 行代码改为：

```
c := s[len(s)]
fmt.Println(len(s), c)
```

此时代码编译不会报错，但是执行时会报错，因为字符串没有这个索引值（从 0 开始算，虽然长度是 7，但最后一字节索引是 6）。

同时，可以注意到第 i 字节并不一定是字符串的第 i 个字符，像汉字这种非 ASCII 字符的 UTF-8 编码由多字节组成。

这个时候再看第 18 行的内容，字符串操作 s[0:7]表示基于 s 字符串的第 0 字节开始到第 7 字节（并不包含第 7 字节）生成一个新字符串。生成的新字符串将包含 7-0 字节。也就是说如果要截取字符串，需要注意并不包含最后那个字节。

1. 连接字符串

除了截取字符串，还可以使用+符号连接字符串：

```
package main

import (
      "fmt"
)

var value1 float64

func main() {
      s := "abcd 你"
      fmt.Println(s[4:] + "好")
      str := "你好, " +
```




仅供非商业用途或交流学习使用


```
            "Go 语言。"
      fmt.Println(str)
      // 由于编译器行尾自动补全分号的缘故，加号必须放在第一行。
}

/*
返回：
你好
你好，Go 语言。
*/
```

在 Go 语言中，字符串是可以使用==和<等符号进行比较的，通过比较逐字节的编码获得结果：

```
package main

import (
      "fmt"
)

var value1 float64

func main() {
      s := "你"
      t := "好"
      if s < t {
            fmt.Println(s[0], t[0])
            fmt.Println(s[1], t[1])
            fmt.Println(s[2], t[2])
      }
      a := "a"
      b := "b"
      if a < b {
            fmt.Println(a[0], "小于", b[0])
      }
}
/*
返回：
228 229
189 165
160 189
97 小于 98
*/
```

注意上面的第 14 行代码，在字符串比较中大小的判断是根据第一字节来判定的，上面的“你好”之所以“你”比“好”小，就是因为“你”的第一字节是 228，“好”的第一字节是 229。可以修改“你好”为“我们”，查看返回结果的不同。

2. 字符串遍历

尽管上面已经尝试过字符串遍历，但是没有解释字符串遍历的两种区别：

```
package main

import "fmt"

func main() {
	s := "我是中国人"
	for i := 0; i < len(s); i++ {
		fmt.Printf("%c", s[i])
		// fmt.Printf("%d = %v\n", i, s[i]) // 输出单字节值
	}
	fmt.Printf("\n")
	for _, v := range s {
		fmt.Printf("%c", v)
	}
	fmt.Print("\n")
}

/*
返回:
æ  æ ¯ä¸-å ½äºº
我是中国人
*/
```

从上面的代码可以看出，第一种方法输出了不可阅读的乱码，这是因为单纯使用 len() 方法是不能遍历字符串的每个字符的，因为在 Go 语言中使用 UTF-8 来保存字符串，所以每个字符可能由多字节组成，只是遍历其中一字节是无法显示为一个完整的字符的，因此遍历字符串一般使用 range 方式，如果要获取各个字符的字节值，可以使用上面注释的语句。

3. 字符串修改

因为在 Go 语言中，字符串内容不能修改，也就不能用 s[i]这种方式修改字符串中的 UTF-8 编码，如果确实要修改，那么可以将字符串的内容复制到另一个可写的变量中，然后再进行修改。一般使用[]byte 或[]rune 类型。

如果要对字符串中的字节进行修改，则转换为[]byte 格式，如果要对字符串中的字符进行修改，则转换为[]rune 格式，转换过程会自动复制数据。

修改字符串中的字节（用[]byte）：

```
package main

import "fmt"
```

```
func main() {
      s := "Hello 世界! "
      b := []byte(s)          // 转换为 []byte, 自动复制数据
      b[5] = ','              // 修改 []byte
      fmt.Printf("%s\n", s)  // s 不能被修改, 内容保持不变
      fmt.Printf("%s\n", b)  // 修改后的数据
}

/*
返回:
Hello 世界!
Hello,世界!
*/
```

修改字符串中的字符（用[]rune）：

```
package main

import "fmt"

func main() {
      s := "Hello 世界! "
      r := []rune(s)          // 转换为 []rune, 自动复制数据
      r[6] = '中'             // 修改 []rune
      r[7] = '国'             // 修改 []rune
      fmt.Println(s)          // s 不能被修改, 内容保持不变
      fmt.Println(string(r)) // 转换为字符串, 又一次复制数据
}

/*
返回:
Hello 世界!
Hello 中国!
*/
```

和 C/C++不一样，Go 语言中的字符串是根据长度限定的，而非特殊字符\0。string 类型的 0 值是长度为 0 的字符串，即空字符串""。

一般的比较运算符（==、!=、<、<=、>=、>）通过在内存中按字节比较来实现字符串的对比。可以通过函数 len()来获取字符串所占的字节长度，例如：len(str)。字符串的内容（纯字节）可以通过标准索引法来获取，在中括号[]内写入索引，索引从 0 开始计数。

- 字符串 str 的第 1 字节：str[0]
- 第 i 字节：str[i - 1]
- 最后 1 字节：str[len(str)-1]



需要注意的是，这种转换方案只对纯 ASCII 码的字符串有效。注意，获取字符串中某字节的地址的行为是非法的，例如：&str[i]。

4. strings 包

对于基本类型来说，字符串所需要执行的操作会比较复杂，所以一般语言都会额外封装一些方法用于处理字符串，Go 语言标准库中也存在这样一个名为 strings 的库。

1）包含判断

判断一个字符串中是否有相应的某个子字符串是经常遇到的一种字符串操作，在 Go 语言中可以使用 strings 包中的两个方法判断，先来看下面的例子：

```
package main

import (
	"fmt"
	"strings"
)

func main() {
	str := "This is an example of a string"
	fmt.Printf("Does the string \"%s\" have prefix %s? \n", str, "Th")
	fmt.Printf("%t\n", strings.HasPrefix(str, "Th"))
	fmt.Printf("Does the string \"%s\" have prefix %s? \n", str, "string")
	fmt.Printf("%t\n", strings.HasSuffix(str, "string"))
	fmt.Printf("Does the string \"%s\" have prefix %s? \n", str, "example")
	fmt.Printf("%t\n", strings.Contains(str, "example"))
}
/*
返回:
Does the string "This is an example of a string" have prefix Th?
true
Does the string "This is an example of a string" have prefix string?
true
Does the string "This is an example of a string" have prefix example?
true
*/
```

代码中使用了两种方法判断字符串包含关系，返回值都是 bool 值。

① 前后缀包含

在上面的代码中，首先使用的是 HasPrefix 这个方法，它用于判断字符串的前缀是否包含指定的子字符串。如上面的例子中，HasPrefix 判断了 str 中开头是否为 Th 字符串，返回的结果是 true；HasSuffix 则相反，用于判断字符串的后缀是否包含指定的子字符串，

其中上面例子中检测了 str 字符串的后缀是否为 string，结果返回 true。

② 子字符串包含

除了检查前后缀，还有字符串中间的包含检测，这时候就需要用到上面代码中出现的 Contains 方法了，它可以判断一个字符串中是否包含指定子字符串。

除此之外还有一个比较常用的包含判断函数，那就是 ContainsAny()：

```
package main

import (
	"fmt"
	"strings"
)

func main() {
	fmt.Println(strings.ContainsAny("team", "i"))
	fmt.Println(strings.ContainsAny("failure", "a & o"))
	fmt.Println(strings.ContainsAny("foo", ""))
	fmt.Println(strings.ContainsAny("", ""))

	fmt.Println(strings.Contains("team", "i"))
	fmt.Println(strings.Contains("failure", "a & o"))
	fmt.Println(strings.Contains("foo", ""))
	fmt.Println(strings.Contains("", ""))
}
/*
返回:
false
true
false    // 注意此处的 false
false

false
false
true    // 注意此处的 true
true
*/
```

与 strings.Contains(str, chars)不同的是，strings.ContainsAny(str, chars)能够匹配更广泛的内容，它可以匹配 Unicode 字符。

2）索引

在 Go 语言中，字符串中的字符都有一个索引值，很多时候我们要操作字符串，就必须获取字符在字符串中的索引值，在 strings 包中 Index 就可以返回指定字符/串的第一个



字符的索引值，如果不存在相应的字符串则返回-1。例如下面的例子：

```
package main

import (
	"fmt"
	"strings"
)

func main() {
	str := "Hi, I'm Job, Hi."

	fmt.Printf("The position of \"Job\" is: ")
	fmt.Printf("%d\n", strings.Index(str, "Job"))

	fmt.Printf("The position of the first instance of \"Hi\" is: ")
	fmt.Printf("%d\n", strings.Index(str, "Hi"))
	fmt.Printf("The position of the last instance of \"Hi\" is: ")
	fmt.Printf("%d\n", strings.LastIndex(str, "Hi"))

	fmt.Printf("The position of \"Tim\" is: ")
	fmt.Printf("%d\n", strings.Index(str, "Burger"))
}

/*
返回:
The position of "Job" is: 8
The position of the first instance of "Hi" is: 0
The position of the last instance of "Hi" is: 13
The position of "Tim" is: -1
*/
```

其中 LastIndex 表示返回字符串最后一个字符出现位置的索引，-1 表示不包含特定字符串。

```
strings.LastIndex(s, str string) int
```

如果是非 ASCII 编码的字符，可以使用 IndexRune 函数来对字符进行定位：

```
strings.IndexRune(s string, ch int) int
// 例如
str := "我是左蓝"
fmt.Printf("The position of \"左\" is: ")
fmt.Printf("%d\n", strings.IndexRune(str, '左'))
/*
返回:
The position of "左" is: 6
*/
```

3）替换

替换字符串最常用的方式其实是通过正则匹配去替换的，灵活度更高，但是 Go 语言对于比较基础的字符串替换已经打包在 strings 包中了，例如 Replace 函数。

```
package main

import (
	"strings"
)

func main() {
	str := "你好世界，这个世界真好。"
	new := "地球"
	old := "世界"
	n := 1
	println(strings.Replace(str, old, new, n))
}
/*
返回：
你好地球，这个世界真好。
*/
```

strings.Replace(str, old, new, n)函数一共有 4 个参数，其中第一个为原字符串，第二个表示原字符串中需要被替换的子字符串，new 是替换内容，n 则表示匹配到第几个 old，如果把上面的 n 改为数字-1，则表示匹配所有。

4）统计

① 出现频率

统计指定字符串出现的频率是常见的字符串操作之一，统计的方法已经包含在 strings 包中，直接调用即可。

```
package main

import (
	"fmt"
	"strings"
)

func main() {
	str := "Golang is cool, right?"
	var manyO = "o"
	fmt.Printf("%d\n", strings.Count(str, manyO))
	fmt.Printf("%d\n", strings.Count(str, "oo"))
}
```

```
/*
返回:
3
1
*/
```

strings.Count(str, manyO)有两个参数，第一个参数是原字符串，第二个参数是指要统计的子字符串。

② 字符数量

除了需要统计字符出现的频率，统计字符串的字符数量也是经常用到的操作之一，在 Go 语言中有很多办法可以获得字符串字符总数，其中比较常见的两种方式如下：

```
package main

import (
	"fmt"
	"unicode/utf8"
)

func main() {
	str := "你好世界"
	fmt.Printf("%d\n", len([]rune(str)))
	fmt.Println(utf8.RuneCountInString(str))
}
```

len([]rune(str))其实是两个方法的组合，先是把字符串通过[]rune 转化，然后统计长度；第二种方法是通过 utf8 包中的 RuneCountInString 函数统计。一般来说更推荐第二种方式，但如果只是单纯统计，没有用到 utf8 的其他函数，倒没必要把整个 utf8 包引入，则可以用第一种方式。

5）大小写转换

ToLower 将字符串中的 Unicode 字符全部转换为相应的小写字符：

```
strings.ToLower(s) string
```

ToUpper 将字符串中的 Unicode 字符全部转换为相应的大写字符：

```
strings.ToUpper(s) string
```

示例：

```
package main

import (
	"fmt"
```

仅供非商业用途或交流学习使用


```
    "strings"
)

var orig = "How are you?"
var lower string
var upper string

func main() {
    fmt.Printf("%s\n", orig)
    lower = strings.ToLower(orig)
    fmt.Printf("%s\n", lower)
    upper = strings.ToUpper(orig)
    fmt.Printf("%s\n", upper)
}
/*
返回:
How are you?
how are you?
HOW ARE YOU?
*/
```

6）修剪

字符串的修剪主要是去掉一些不需要的字符，一般可以使用 strings.Trim 函数来修剪。注意下面函数中第二个参数的变化以及修剪的结果：

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    fmt.Println("## Trim 函数的用法")
    fmt.Printf("%q\n", strings.Trim(" !!! Golang !!! ", "! "))
    fmt.Printf("%q\n", strings.Trim(" !!! Golang !!! ", " ! "))
    fmt.Printf("%q\n", strings.Trim(" !!! Golang !!! ", "!"))

    fmt.Println("## TrimLeft 函数的用法")
    fmt.Printf("%q\n", strings.TrimLeft(" !!! Golang !!! ", "! "))
    fmt.Printf("%q\n", strings.TrimLeft(" !!! Golang !!! ", " ! "))
    fmt.Printf("%q\n", strings.TrimLeft(" !!! Golang !!! ", "!"))

    fmt.Println("## TrimSpace 函数的用法")
    fmt.Println(strings.TrimSpace(" \t\n 这是\t 一句话 \n\t\r\n"))
}
/*
```

仅供非商业用途或交流学习使用


```
返回:
## Trim 函数的用法
"Golang"
"Golang"
" !!! Golang !!! "
## TrimLeft 函数的用法
"Golang !!! "
"Golang !!! "
" !!! Golang !!! "
## TrimSpace 函数的用法
这是　一句话
*/
```

Trim 这个函数修剪的是字符串开头或者结尾的字符，也就是说不匹配中间的字符，而且 Trim 函数第二个参数虽然类型是 string，但实际上替换时是把字符串转换成 rune 后操作的，例如：

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    fmt.Printf("%q\n", strings.Trim("今天天气真好", "今"))
    fmt.Printf("%q\n", strings.Trim("今天天气真好", "天"))
    fmt.Printf("%q\n", strings.Trim("今天天气真好", "今天"))
}
/*
返回:
"天天气真好"
"今天天气真好"
"气真好"
*/
```

第一句的确裁剪了“今”字，但是第二句的“天”并非字符串首尾，所以不会裁剪，重点在于第三句，可以看到后面的“天”全部被裁剪了，这是因为 Trim 函数把第二个参数转换为了 rune 类型，进而逐一匹配替换——换句话说，实际上“今天”是被分为两个字符分别匹配裁剪的，所以只返回了“气真好”。

7）分割

分割相对来说比较简单，Go 语言的 strings 包中封装了分割函数 strings.Split，函数返回的是一个切片（slice）。

仅供非商业用途或交流学习使用


```
package main

import (
	"fmt"
	"strings"
)

func main() {
	fmt.Printf("%q\n", strings.Split("a,b,c", ","))
	fmt.Printf("%q\n", strings.Split("a boy a girl a dog a cat", "a "))
	fmt.Printf("%q\n", strings.Split("xyz ", ""))
}
/*
返回:
["a" "b" "c"]
["" "boy " "girl " "dog " "cat"]
["x" "y" "z" " "]
*/
```

8）插入字符

strings.Join 用于将元素类型为 string 的 slice 使用分隔符号拼接组成一个字符串。

```
package main

import (
	"fmt"
	"strings"
)

func main() {
	str := "The quick brown fox jumps over the lazy dog 中文"
	strSli := strings.Fields(str)
	fmt.Printf("%s\n", strSli)
	for _, val := range strSli {
		fmt.Printf("%s ", val)
	}
	fmt.Println()
	str2 := strings.Join(strSli, ";")
	fmt.Printf("%s\n", str2)
}
/*
返回:
[The quick brown fox jumps over the lazy dog 中文]
The quick brown fox jumps over the lazy dog 中文
The;quick;brown;fox;jumps;over;the;lazy;dog;中文
*/
```

strings.Fields 函数用于把字符串转换为字符串切片，然后通过 range 获得每个切片值，

仅供非商业用途或交流学习使用

最后使用 strings.Join 向字符串插入指定的字符。除此之外还可以使用字节缓冲 bytes.Buffer 连接字符串。

5. strconv 包

这个包主要用于字符串与其他类型的转换，strconv 通过包内的变量获取程序运行的操作系统平台下 int 类型所占的位数（strconv.IntSize），一般来说，几乎所有类型都可以被转换为字符串，但反过来要把字符串转换成其他类型就不一定能够成功了。前面我们看过了 rune 和 string 类型的转换，现在来看数字与字符串之间的转换。

从数字类型转换到字符串，Go 语言提供了 strconv.Itoa 和 strconv.FormatFloat()等函数，其中 Itoa 是把十进制数转换为字符串，而 FormatFloat 则是将 64 位浮点型的数字转换为字符串，函数参数如下：

```
strconv.FormatFloat(f float64, fmt byte, prec int, bitSize int) string
```

其中 fmt 表示格式（其值可以是'b'、'e'、'f'或'g'），prec 表示精度，bitSize 的值为 32 则表示 float32，为 64 则表示 float64。

反过来从字符串到数字类型，Go 语言也提供了一些函数，例如 strconv.Atoi(s string) (i int, err error)将字符串转换为 int 型，strconv.ParseFloat(s string, bitSize int) (f float64, err error)将字符串转换为 float64 型。这些函数会返回两个值，第一个是转换后的结果（如果转换成功），第二个是可能出现的错误。

对于多返回值的函数，一般可以把语句写成下面这样：

```
package main

import (
	"fmt"
	"strconv"
)

func main() {
	orig := "233"
	fmt.Printf("orig 当前是%T 类型，且操作系统是%d 位。\n", orig, strconv.IntSize)

	num, err := strconv.Atoi(orig) // num, _ := strconv.Atoi(orig)
	fmt.Printf("num 当前是%T 类型，且数值是%d。\n", num, num)
	fmt.Printf("%s\n", err)
	num += 5
	newS := strconv.Itoa(num)
	fmt.Printf("%s\n", newS)
```

仅供非商业用途或交流学习使用


```
}
/*
返回:
orig 当前是 string 类型，且操作系统是 64 位。
num 当前是 int 类型，且数值是 233。
%!s(<nil>)
238
*/
```

尝试把数字改成字母，程序会输出 strconv.Atoi: parsing "jhgf": invalid syntax，表示无法转换。

3.3.3 字符串格式化

前面的内容中经常出现 T%、d%等符号，关于这些符号的含义总结在表 3-3 中。

表 3-3 格式化指令含义

格式化指令	含　义
%%	%字面量
%b	一个二进制整数，将一个整数格式化为二进制的表达方式
%c	一个 Unicode 的字符
%d	十进制数值
%o	八进制数值
%x	小写的十六进制数值
%X	大写的十六进制数值
%U	一个 Unicode 表示法表示的整型码值，默认是 4 个数字字符
%s	输出以原生的 UTF-8 字节表示的字符，如果 console 不支持 UTF-8 编码，则会输出乱码
%t	以 true 或者 false 的方式输出布尔值
%v	使用默认格式输出值，或者如果方法存在，则使用类型的 String()方法输出的自定义值
%T	输出值的类型

在 Go 语言中单个字符可以使用单引号（'）来创建，字符串支持切片操作，但是需要注意的是，如果字符串都是由 ASCII 字符组成的，那可以随便使用切片进行操作，但是如果字符串中包含其他非 ASCII 字符，直接使用切片获取想要的单个字符时应十分小心，因为对字符串直接使用切片时是通过字节进行索引的，但是非 ASCII 字符在内存中可能不是由一字节组成的。如果想对字符串中的字符依次访问，可以使用 range 操作符。

仅供非商业用途或交流学习使用

3.4　布尔型

真或假，布尔型的值只可以是 true 或者 false。一个简单的例子：var b bool = true。

3.4.1　布尔型的表示

Go 语言中的布尔类型和其他编程语言基本一致，关键字为 bool，可赋值为 true 或者 false，默认为 false。例如：

```
// 直接声明为 bool 类型
v := true
// 也可以让 Go 语言编译器推导为 bool 类型
x := (1 == 2)
```

布尔类型无法被其他类型赋值，也不支持类型转换。Go 语言只有 true 和 false 两个值，不支持使用 0 和 1 表示真假。

```
var b bool
b = 1 // 编译错误，应该为 b = false
b = bool(1) // 编译错误，应该为 b = bool(false)
```

可以通过推导的方式让编译器在编译或者程序运行时推导布尔变量的值，只要推导表达式中两个值的类型相同就可以使用相等（==）或者不等（!=）等运算符进行比较，并返回一个布尔型的值。

```
package main

import (
	"fmt"
	"os"
)

// USER 用户名称
var USER = os.Getenv("USER")

func main() {
	var b bool
	b = (1 != 0)          // 编译时推断变量的值
	u := ("user" == USER) // 运行时根据环境判断变量的值
	fmt.Println(u, b)
}

// 返回 true
```

当相等运算符两边的值完全相同（类型相同、值相同）时，便会返回 true，否则返回

仅供非商业用途或交流学习使用

false。如果值的类型是接口，它们也必须都实现了相同的接口才能比较。如果类型不同，那么必须经过类型转换，使得两者类型相同之后才可以进行比较，类型转换的方法在本章稍后会有介绍。

3.4.2 布尔型的运算

布尔型的常量和变量也可以通过逻辑运算符（非!、和&&、或||）结合来产生另外一个布尔值，这样的逻辑语句就其本身而言，并不是一个完整的 Go 语言语句。

逻辑值可以被用于条件结构中的条件语句（第 5 章），以便测试某个条件是否满足。另外，和&&、或||与相等==或不等!=属于二元运算符，而非!属于一元运算符。在接下来的内容中，会使用 T 来代表条件符合的语句，用 F 来代表条件不符合的语句。

Go 语言中包含以下逻辑运算符。

非运算符：!

```
!T -> false
!F -> true
```

非运算符用于取得和布尔值相反的结果。

和运算符：&&

```
T && T -> true
T && F -> false
F && T -> false
F && F -> false
```

只有当两边的值都为 true 时，和运算符的结果才是 true。

或运算符：||

```
T || T -> true
T || F -> true
F || T -> true
F || F -> false
```

只有当两边的值都为 false 的时候，或运算符的结果才是 false，其中任意一边的值为 true 就能够使得该表达式的结果为 true。

在 Go 语言中，&&和||是具有快捷性质的运算符，当运算符左边表达式的值已经能够决定整个表达式的值时（&&左边的值为 false，||左边的值为 true），运算符右边的表达式将不会被执行。利用这个性质，如果你有多个条件判断，应当将计算过程较为复杂的表达

仅供非商业用途或交流学习使用

式放在运算符的右侧，以减少不必要的运算。

利用括号同样可以升级某个表达式的运算优先级。在格式化输出时，可以使用%t 来表示要输出的值为布尔型。布尔值（以及任何结果为布尔值的表达式）最常用在条件结构的条件语句中，例如第 4 章的顺序编程中的逻辑判断。

对于布尔值的好的命名能够很好地提升代码的可读性，例如以 is 或者 Is 开头的 isSorted、isFinished 等，使用这样的命名能够在阅读代码时获得与阅读正常语句一样的良好体验，标准库中的命名（unicode.IsDigit(ch)等）也是遵循这个规则的。

3.5　基本数据类型的扩展

除了上面介绍到的几种基本类型，Go 语言还有更复杂的数据类型，这些将会在后续章节中介绍，在本节中，还需要进一步认识常见的数据类型操作，例如数据类型转换、自定义类型、类型别名等，还有一个比较有趣的概念——指针，这些内容对后续学习都至关重要。

3.5.1　强制类型转换

类型转换用于将一种数据类型的变量转换为另外一种类型的变量。Go 语言类型转换基本格式如下：

```
type_name(expression)
```

在做强制类型转换时，需要注意数据长度被截短而发生的数据精度损失（比如将浮点数强制转为整数）与值溢出（值超过转换的目标类型的值范围时）问题。

以下例子将整型转化为浮点型并计算结果，将结果赋值给浮点型变量：

```
package main

import "fmt"

func main() {
      sum := 11
      count := 3
      mean := float32(sum) / float32(count)
      mean2 := sum / count
      fmt.Printf("mean 的值为: %f\n", mean)
      fmt.Printf("mean2 的值为: %d\n", mean2)
}
```

仅供非商业用途或交流学习使用


```
/*
返回:
mean 的值为: 3.666667
mean2 的值为: 3
*/
```

3.5.2 自定义类型

在 Go 语言中，自定义类型实际上与结构体有关，本节暂时不会介绍结构体相关的内容。

```
type stu struct {
      Name string /* 首字母大写，允许其他包直接使用，可以直接使用 stu.Name = 'Huang' 也
可以使用 setName 和 getName */
      age  int    // 不允许外面的包使用，可以使用 setAge 和 getAge 方法
}
```

关于结构体的内容详见后面章节。

3.5.3 类型别名

我们知道 byte 类型实际上是 int8 类型的别名，而 rune 类型实际上是 int32 类型的别名，对于开发者而言，Go 语言也允许开发者自定义类型别名。

```
package main

import (
      "fmt"
)

type (
      字符串  string
)

func main() {
      var b 字符串
      b = "这是中文。"
      fmt.Println(b)
      a := "这也是中文。"
      // fmt.Println(b + a)
      fmt.Println(string(b) + a)
}

/*
返回:
这是中文。
这是中文。这也是中文。
*/
```

在上面的例子中，注意注释的那一句，在编译器中这种写法是报错的，因为类型别名与原类型是两种类型，不能直接操作，需要进行类型转换。

3.5.4　指针

众所周知所有指针变量都是一个内存位置，每个内存位置都有其定义的地址，可以使用&运算符来访问它，这个运算符表示内存中的地址。例如打印变量 a 的地址：

```
package main

import "fmt"

func main() {
        a := 10
        fmt.Printf("%x\n", &a)
}
/*
返回：
c4200160c8
*/
```

知道什么是内存地址以及如何访问它之后，再来看指针的概念。指针是一个变量，其值是另一个变量的地址，即存储器位置的直接地址。类似变量或常量一样，必须先声明一个指针，然后才能使用它来存储任何变量地址。指针变量声明的一般形式是：

```
var name *type
```

所有指针的值的实际数据类型（无论是整数、浮点数还是其他数据类型）都是相同的，它表示内存地址的长十六进制数。

使用指针基本上是三个步骤：定义一个指针变量，将一个变量的地址赋值给一个指针，最后访问指针变量中可用地址的值。下面是一个例子：

```
package main

import "fmt"

func main() {
        a := 20
        ap := &a
        fmt.Printf("a 的地址：%x\n", &a)
        fmt.Printf("ap 的地址：%x\n", ap)
        fmt.Printf("*ap 的值：%d\n", *ap)
}
```

```
/*
返回:
a 的地址: c4200160c8
ip 的地址: c4200160c8
*ip 的值: 20
*/
```

Go 语言中通过使用一元运算符*来取得指向指针存储的内存地址所对应的值（指针的格式化标识符为%p）。Go 语言指针不支持指针运算，也不支持->运算符，可以直接用.访问目标成员（涉及 struct，以后会介绍）。

1. nil 指针

Go 语言编译器为指针变量分配一个 nil 值，以防指针没有确切的地址分配，这是在变量声明的时候完成的。指定为 nil 值的指针称为 nil 指针。nil 指针是在几个标准库中定义的值为零的常量。例如：

```
package main

import "fmt"

func main() {
      var ptr *int
      fmt.Printf("ptr 的值是: %x\n", ptr)
}

/*
返回:
ptr 的值是: 0
*/
```

在大多数操作系统中，程序不允许访问地址 0 处的内存，因为该内存是由操作系统保留的。然而，存储器地址 0 具有特殊意义，它表示指针不打算指向可访问的存储器位置。但是按照惯例，如果指针包含 nil 值，则假设它不指向任何对象。要检查是否为 nil 指针，可以使用 if 语句，如下所示：

```
if(ptr != nil)
if(ptr == nil)
```

对一个空指针的反向引用是不合法的，并且会使程序崩溃：

```
var p *int = nil; *p = 10
```

2. 指针的指针

指针的指针声明方式如下：

```
package main

import "fmt"

func main() {
      var a *int // 定义一个 a 指针，指向 nil
      aP := &a   // 空指针 a 的指针
      fmt.Printf("a-->nil: %x\n", a)
      fmt.Printf("aP-->a: %x\n", aP)
      fmt.Printf("aP-->a-->nil（指针 aP 指向的指针 a 的内存地址）: %x\n", *aP)
      fmt.Printf("&aP-->aP(表示 aP 在内存中的地址): %x\n", &aP)
}

/*
返回:
a-->nil: 0
aP-->a: c42000c028
aP-->a-->nil（指针 aP 指向的指针 a 的内存地址）: 0
&aP-->aP(表示 aP 在内存中的地址): c42000c030
*/
```

如果要先定义一个指针的指针，可以这样定义：var ptr **int;，如下面所示：

```
package main

import "fmt"

func main() {
      a := 10
      aP := &a
      aPP := &aP
      fmt.Printf("a: %d\n", a)
      fmt.Printf("aP: %x\n", aP)
      fmt.Printf("*aP: %d\n", *aP)
      fmt.Printf("aPP: %x\n", aPP)
      fmt.Printf("*aPP: %x\n", *aPP)
      fmt.Printf("**aPP: %d\n", **aPP)
}

/*
返回:
a: 10
aP: c4200160c8
*aP: 10
aPP: c42000c028
*aPP: c4200160c8
**aPP: 10
*/
```

3. 指针数组

数组的概念虽然还没解释，不过如果接触过其他编程语言，想必对数组不陌生，数组中通常有多个值，所以指向数组的指针也会有多个值——确切地说是指针数组有多个值，例如下面代码：

```
package main

import "fmt"

// MAX 数组大小
const MAX int = 3

func main() {
      a := []int{10, 100, 200}
      var ptr [MAX]*int

      for i := 0; i < MAX; i++ {
            ptr[i] = &a[i]
            fmt.Printf("a[%d]的地址: %x\n", i, ptr[i])
      }

      for i := 0; i < MAX; i++ {
            fmt.Printf("a[%d]的值是: %d\n", i, *ptr[i])
      }
}
/*
返回:
a[0]的地址: c420012480
a[1]的地址: c420012488
a[2]的地址: c420012490
a[0]的值是: 10
a[1]的值是: 100
a[2]的值是: 200
*/
```

这里将 ptr 声明为一个指针数组，数组长度为 MAX。因此，ptr 中的每个元素现在保存一个指向 int 值的指针。示例使用三个整数，它们将存储在指针数组中。

4. 传递给函数

Go 语言允许传递指针到函数中，只需将函数的参数声明为指针类型。虽然还没到函数的章节，但是不妨碍理解，如下面代码传递了两个指针给一个函数，并在函数里改变它们的值，这个值反映在调用函数中：

```
package main

import "fmt"

func main() {
	a := 100
	b := 200

	fmt.Printf("交换之前 a 的值为：%d\n", a)
	fmt.Printf("交换之前 b 的值为：%d\n", b)

	// 把 a 和 b 的地址传递给函数，然后 a 和 b 的地址交换，最终实现两个值的交换
	swap(&a, &b)

	fmt.Printf("交换之后 a 的值为：%d\n", a)
	fmt.Printf("交换之后 b 的值为：%d\n", b)
}

func swap(x *int, y *int) {
	var temp int
	temp = *x
	*x = *y
	*y = temp
}
/*
返回：
交换之前 a 的值为：100
交换之前 b 的值为：200
交换之后 a 的值为：200
交换之后 b 的值为：100
*/
```

3.6　本章小结

本章介绍了 Go 语言的基础类型，Go 语言将数据类型分为四类：基础类型、复合类型、引用类型和接口类型。本章介绍的基础类型包括：数字、字符串和布尔型。虽然数据种类很多，但它们都是对程序中一个变量或状态的间接引用，这意味着对任一引用类型数据的修改都会影响该引用的所有副本，关于值引用和间接引用会在后面章节深入探讨。

第4章 流程控制

流程控制是顺序编程中必不可少的一部分，它是整个编程基础的重要一环。在顺序编程的流程控制部分，Go 语言和其他主流语言有一些差别，主要体现在 Go 语言没有 do-while 语句，因此 for 语句拥有更广泛的含义与用途。另一方面 switch 语句也有一些扩展，例如支持类型判断和初始化子语句等。

除了这些常见的流程控制语法的关键字，Go 语言还有三个特殊的关键字，分别是 defer——用于捕获异常和资源回收等工作；select——用于多分支选择（配合通道使用）；go——用于异步启动 goroutine 并执行特定函数。

4.1 条件语句

我们一般把一对大括号中的代码称为一个代码块，例如一个函数，或者一个 if 语句等都是一个代码块，每个代码块内部可以拥有自己的局部变量。

在 Go 语言中，一个源代码文件（.go）本身就是一个代码块，一个代码包也是一个代码块，甚至更广泛而言，整个项目的代码都是一个代码块。由此可知，代码块是互相包含的关系，往小了说，像本章的流程控制语句中的 if、for、switch 和 select 等语句括起来的代码也是代码块，甚至更小的——每一个 case 分支都是一个代码块。

之所以要解释代码块的概念，是为了方便理解作用域，在 Go 语言中的预定义标识符作用域是整个项目，而一个源代码文件中定义的实体（常量、变量、类型等）作用域就是当前代码块（包导入可以扩大作用域），函数之中定义的实体（常量、变量、类型等）作用域就是函数所在的代码块。

if 语句是所有流程控制语句中最常用的一个，这个语句可以根据条件表达式来执行代码块中的两个分支，表达式只能返回布尔类型，即 true 或 false，当返回值为 true 时执行 if 后面的代码块，当返回值为 false 时，执行 else 后面的代码块。

4.1.1　if 判断

例如最简单的判断：

```
package main

import "fmt"

func main() {
	a := 1		// 定义一个变量 a，赋值为 1
	if a < 20 { // 判断 a 是否小于 20
		fmt.Printf("a 小于 20\n")
	}
	fmt.Printf("a 的值是：%d\n", a)
}
/*
保存文件为 if.go，然后使用 go run if.go 运行程序
返回：
a 小于 20
a 的值是：1
*/
```

在上面的判断中，因为 a 的值为 1，小于 20，表达式返回 true，所以可以返回“a 小于 20”的信息，但如果把 a 的值设置为 100，那么程序判断语句将返回 false，判断语句也就没有任何信息输出，为了增加 false 的返回情况就需要用到 if-else 语句。

4.1.2　if-else 判断

在上面的判断中加入一个 else 代码块即可，例如：

```
package main

import "fmt"

func main() {
	a := 100	// 定义一个变量 a，赋值为 100
	if a < 20 { // 判断 a 是否小于 20
		fmt.Printf("a 小于 20\n")
	} else {
		fmt.Printf("a 大于 20\n")
```

```
        }
        fmt.Printf("a 的值是：%d\n", a)
}
/*
保存文件为 if-else.go，然后使用 go run if-else.go 运行程序
返回：
a 大于 20
a 的值是：100
*/
```

改变其中 a 的值，例如 10，可以看程序输出的变化。但是现在，如果想判断 a 的值是否小于 20 且大于 10，那么当前语句就无法满足要求了。

可以在 if-else 语句中进行修改：

```
package main

import "fmt"

func main() {
        a := 11     // 定义一个变量 a，赋值为 11
        if a < 20 { // 判断 a 是否小于 20
                fmt.Printf("a 小于 20\n")

                if a > 10 { // 判断 a 是否大于 10
                        fmt.Printf("a 大于 10\n")
                } else {
                        fmt.Printf("a 小于 10\n")
                }
        } else {
                fmt.Printf("a 大于 20\n")
        }
        fmt.Printf("a 的值是：%d\n", a)
}

/*
保存文件为 if-else.go，然后使用 go run if-else.go 运行程序
返回：
a 小于 20
a 大于 10
a 的值是：11
*/
```

上面代码的确准确判断了 a 的值，但是嵌套了一层 if-else 语句，这在编程规范中可以说是大忌，特别是在逻辑复杂的情况下，嵌套 if 语句将非常影响性能，因此就有了 else-if 语句。

4.1.3　else-if 判断

else-if 语句是在前面 if-else 语句之上再度扩展的，为了解决多重判断的问题，例如：

```
package main

import "fmt"

func main() {
	a := 11
	if a > 20 {
		fmt.Printf("a 大于 20\n")
	} else if a < 10 {
		fmt.Printf("a 小于 10\n")
	} else {
		fmt.Printf("a 大于 10\n")
		fmt.Printf("a 小于 20\n")
	}
	fmt.Printf("a 的值是: %d\n", a)
}

/*
保存文件为 else-if.go，然后使用 go run else-if.go 运行程序
返回:
a 大于 10
a 小于 20
a 的值是: 11
*/
```

else-if 语句可以连续使用多个 else if 关键字，例如判断一个数字是否大于 10 小于 20 且不等于 11：

```
package main

import "fmt"

func main() {
	a := 13
	if a > 20 {
		fmt.Printf("a 大于 20\n")
	} else if a < 10 {
		fmt.Printf("a 小于 10\n")
	} else if a == 11 {
		fmt.Printf("a 等于 11\n")
	} else {
		fmt.Printf("a 大于 10\n")
		fmt.Printf("a 小于 20\n")
		fmt.Printf("a 不等于 11\n")
```

```
        }
        fmt.Printf("a 的值是: %d\n", a)
}

/*
保存文件为 else-if.go，然后使用 go run else-if.go 运行程序
返回:
a 大于 10
a 小于 20
a 不等于 11
a 的值是: 13
*/
```

上面使用了两个 else-if 语句，通过修改 a 的值来体验这个语法的特点，例如修改 a 的值为 11，则返回“a 等于 11”的信息。

4.1.4 初始化子语句

上面说过 if 语句可以有一个子语句，用于初始化局部变量，例如：

```
package main

import "fmt"

func main() {
        if a := 10; a < 20 {
                fmt.Printf("a 小于 20\n")
        } else {
                fmt.Printf("a 的值是: %d\n", a)
        }
}
```

上面例子中，在 if 关键字后面有一句 a := 10 子语句，表示初始化 a 的值。子语句只能有一个表达式，即下面这样是无法编译的：

```
func main() {
        if b:= 10; a := 10; a < 20 {  // 编译出错，初始化子语句只能有一个表达式
                fmt.Printf("a 小于 20\n")
        } else {
                fmt.Printf("a 的值是: %d\n", a)
        }
}
```

除此之外，因为 a 的值是在 if 代码块中定义的，所以不能在代码块之外调用，例如下面代码也是无法编译的：

仅供非商业用途或交流学习使用


```
package main

import "fmt"

func main() {
      if a := 10; a < 20 {
            fmt.Printf("a 小于 20\n")
      }
      fmt.Printf("a 的值是: %d\n", a)  /* 编译出错，a 是在 if 代码块中定义的，所以不能在函
                                          数中调用 */
}
```

所有条件语句都是从上到下依次执行的，遇到 true 就执行 true 分支的操作，否则执行 else 中的操作，换句话说，只要遇到一个 true 返回值，后面的表达式都不会被执行。

4.2　选择语句

在上面使用 if 条件语句时，一般用于二元判断，对于多元判断使用条件语句就会显得烦琐，所以就有了选择语句。例如判断一个数字属于哪个区间就可以用选择语句轻松实现，避免条件语句烦琐的嵌套过程。

4.2.1　switch 语句

在 Go 语言中，switch 表示选择语句的关键字，switch 语句会根据初始化表达式得出一个值，然后根据 case 语句的条件，执行相应的代码块，最终返回特定内容。每个 case 被称为一种情况，只有当初始化语句的值符合 case 的条件语句时，case 才会被执行。

如果没有遇到符合的 case，则可以使用默认的 case（default case），如果已经遇到了符合的 case，那么后面的 case 都不会被执行。

与其他编程语言不同的是，在 Go 语言编程中，switch 有两种类型。

- 表达式 switch：在表达式 switch 中，case 包含与 switch 表达式的值进行比较的表达式。
- 类型 switch：在类型 switch 中，case 包含与特殊注释的 switch 表达式的类型进行比较的类型。

1. 表达式 switch

下面是表达式 switch 的一个简单例子：

仅供非商业用途或交流学习使用


```
package main

import "fmt"

func main() {
	grade := "B"
	marks := 90

	switch marks {
	case 90:
		grade = "A"
	case 80:
		grade = "B"
	case 60, 70:
		grade = "C"
	default:
		grade = "D"
	}
	fmt.Printf("你的成绩为%s\n", grade)
}
/*
返回:
你的成绩为 A
*/
```

在上面的例子中，switch 后面是一个变量值，选择语句根据这个变量值判断执行哪个 case，在上面例子中 marks 为 90，所以 grade 为 A。如果把 marks 改为 40，那么从上到下，依次判断均没有符合的 case，于是执行 default 中的情况，即输出 grade 为 D。

这种方式虽然简单，但一般不这样写，因为不容易扩展这个选择语句，例如把分数改为 100 或者 91 时，也会返回“你的成绩为 D”，这明显不符合实际情况。这是因为选择语句中没有考虑值的范围的问题，现在假设 90 分到 100 分为 A，80 分到 89 分为 B，60 分到 79 分为 C，低于 60 分则为 D。这种情况下上面的单值判断已经不能满足我们此时的需求。

因此就需要完整的 switch 表达式写法了：

```
package main

import "fmt"

func main() {
	grade := "E"
	marks := 90
```

仅供非商业用途或交流学习使用


```
    switch {
    case marks >= 90:
        grade = "A"
    case marks >= 80:
        grade = "B"
    case marks >= 70:
        grade = "C"
    case marks >= 60:
        grade = "D"
    default:
        grade = "E"
    }

    switch {
    case grade == "A":
        fmt.Printf("成绩优秀! \n")
    case grade == "B":
        fmt.Printf("表现良好! \n")
    case grade == "C", grade == "D": // case表达式可以有多个
        fmt.Printf("再接再厉! \n")
    default:
        fmt.Printf("成绩不合格! \n")
    }
    fmt.Printf("你的成绩为%s\n", grade)
}
/*
返回:
成绩优秀!
你的成绩为 A
*/
```

上面例子中有两个 switch 相互关联，第一个 switch 语句判断成绩所属区间，并得出 grade 的值，然后第二个 switch 根据 grade 的值返回相应的话语。

注意，每一个 case 都可以拥有多个表达式，它的含义与 fallthrough 一样，例如下面两种写法是一样的意思：

```
/* 多表达式写法 */
    case grade == "C", grade == "D": // case表达式可以有多个
        fmt.Printf("再接再厉! \n")
/* fallthrough写法 */
    case grade == "C":
        fallthrough
    case grade == "D":
        fmt.Printf("再接再厉! \n")
```

fallthrough 关键词可以把当前 case 控制权交给下一个 case 语句判断。

仅供非商业用途或交流学习使用

2. 类型 switch

类型 switch 语句针对变量类型判断执行哪个 case 代码块，下面是一个简单的例子：

```
package main

import "fmt"

var x interface{} // 空接口

func main() {
      x = 1  // 修改 x 的值，查看返回结果的变化
      switch i := x.(type) {  // 这里表达式只有一句初始化子语句
      case nil:
            fmt.Printf("这里是 nil，x 的类型是%T", i)
      case int:
            fmt.Printf("这里是 int，x 的类型是%T", i)
      case float64:
            fmt.Printf("这里是 float64，x 的类型是%T", i)
      case bool:
            fmt.Printf("这里是 bool，x 的类型是%T", i)
      case string:
            fmt.Printf("这里是 string，x 的类型是%T", i)
      default:
            fmt.Printf("未知类型")
      }
}
/*
返回：
这里是 int，x 的类型是 int
*/
```

类型 switch 的初始化子语句中需要判断的变量必须是具有接口类型的变量(如果是固定类型的变量就没有判断的意义了)。在语法上类型 switch 与表达式 switch 没有太大区别。

4.2.2 switch 初始化语句

switch 语句同样拥有初始化子语句，和 if 一样均是写在关键字后面，只能有一句语句，例如：

```
/* 单值判断写法 */
      switch marks := 90; marks {
      case 90:
            grade = "A"
      case 80:
            grade = "B"
```

仅供非商业用途或交流学习使用


```
        case 70:
                grade = "C"
        case 60:
                grade = "D"
        default:
                grade = "E"
        }
/* 范围表达式写法 */
        switch marks := 90; {  // 分号不能省略
        case marks >= 90:
                grade = "A"
        case marks >= 80:
                grade = "B"
        case marks >= 70:
                grade = "C"
        case marks >= 60:
                grade = "D"
        default:
                grade = "E"
        }
```

在 Go 语言中同样有 break 语句，不过 Go 语言的 break 语句比较特殊，我们会在稍后的标签中介绍 break 语句的用法。

4.2.3　select 语句

在 Go 语言中，除了 switch 语句，还有一种选择语句——select，这种选择语句用于配合通道（channel）的读写操作，用于多个 channel 的并发读写操作。

由于本章还未介绍并发概念，所以此处不会介绍过多后面的知识，只需记住 select 的用途即可，switch 是按顺序从上到下依次执行的，而 select 是随机选择一个 case 来判断，直到匹配其中的一个 case，举个例子：

```
package main

import (
        "fmt"
)

func main() {
        a := make(chan int, 1024)
        b := make(chan int, 1024)

        for i := 0; i < 10; i++ {
                fmt.Printf("第%d 次, ", i)
```

仅供非商业用途或交流学习使用


```
            a <- 1
            b <- 1

            select {
            case <-a:
                    fmt.Println("from a")
            case <-b:
                    fmt.Println("from b")
            }
      }
}

/*
返回（每次返回结果不同）：
第 0 次，from a
第 1 次，from b
第 2 次，from a
第 3 次，from a
第 4 次，from b
第 5 次，from b
第 6 次，from b
第 7 次，from a
第 8 次，from a
第 9 次，from a
*/
```

上述代码的意思是，同时在 a 和 b 中选择，哪个有内容就从哪个读，由于 channel 的读写操作是阻塞操作，使用 select 语句可以避免单个 channel 的阻塞。此外 select 同样可以使用 default 代码块，用于避免所有 channel 同时阻塞。具体内容可见并发编程。

4.3 循环语句

循环语句是编程中常使用的流程控制语句之一，在 Go 语言中，循环语句的关键字是 for，没有 while 关键字。for 语句可以根据指定的条件重复执行其内部的代码块，这个判断条件一般是由 for 关键字后面的子语句给出的。

例如一个简单的 for 循环例子：

```
package main

import "fmt"

func main() {
      for a := 0; a < 5; a++ {
```

仅供非商业用途或交流学习使用


```
            fmt.Printf("a 的值是: %d\n", a)
        }
}
/*
返回:
a 的值是: 0
a 的值是: 1
a 的值是: 2
a 的值是: 3
a 的值是: 4
*/
```

上面 for 关键字后面有三个子语句，初始化变量 a 为 0，并判断当 a 小于 5 时执行下面代码块的内容，每次判断 a 的值都加 1，直到不符合初始化语句的判断条件，进而退出循环。

4.3.1　for 的子语句

for 语句后面的三个子语句我们称为：初始化子语句、条件子语句和后置子语句，这三者不能颠倒顺序，其中条件子语句是必需的，条件子语句会返回一个布尔型，true 则执行代码块，false 则跳出循环。例如以下示例代码中就省略了初始化子语句和后置子语句：

```
package main

import "fmt"

func main() {
        a := 0
        b := 5
        for a < b {
                a++
                fmt.Printf("a 的值是: %d\n", a)
        }
}
/*
返回:
a 的值是: 0
a 的值是: 1
a 的值是: 2
a 的值是: 3
a 的值是: 4
*/
```

在上面的例子中，for 关键字后面只有一个 a < b 的判断语句，这是典型的条件判断语句，它实际上是三个子语句的简写：

```
for ; a < b ;
// Go 语言编译器会自动判断三个子语句中是否存在条件子语句

// 例如写成这样就会报错
for ; ; a < b
```

后置子语句的意思就是先进行条件子语句判断，for 代码块执行之后再对条件变量操作的语句进行判断，上面例子中的 a++就是一个后置子语句。

4.3.2 range 子语句

每一个 for 语句都可以使用一个特殊的 range 子语句，其作用类似于迭代器，用于轮询数组或者切片值中的每一个元素，也可以用于轮询字符串的每一个字符，以及字典值中的每个键值对，甚至还可以持续读取一个通道类型值中的元素。

例如：

```
package main

import "fmt"

func main() {
	str := "abcz"
	for i, char := range str {
		fmt.Printf("字符串第%d 个字符的值为%d\n", i, char)
	}
	for _, char := range str { // 忽略第一个值（忽略 index）
		println(char)
	}
	for i := range str { // 忽略第二个值
		fmt.Println(i)
	}
	for range str { // 忽略全部返回值，只执行下面代码块
		println("执行成功")
	}
}
/*
返回：
字符串第 0 个字符的值为 97
字符串第 1 个字符的值为 98
字符串第 2 个字符的值为 99
字符串第 3 个字符的值为 122
0
1
2
3
```

```
97
98
99
122
执行成功
执行成功
执行成功
执行成功
*/
```

range 关键字右边是 range 表达式，表达式一般写在 for 语句前面，以便提高代码易读性。像上面的例子可以写成：

```
for i := range "abcz"
// 把原来的 str := "abcz" 表达式写到 range 关键字后面
```

这样不仅降低了可读性，也不容易管理后续的循环代码块。

range 关键字左边表示的是一对索引-值对，根据不同的表达式返回不同的结果，详见表 4-1。

表 4-1　range 返回值的输出

右边表达式返回的类型	第一个值	第二个值
string	index	str[index]，返回类型为 rune
array/slice	index	str[index]
map	key	m[key]
channel	element	

从表 4-1 中可以看出，除了轮询字符串，还支持其他类型，例如数组、切片、字典甚至通道等：

```
package main

import "fmt"

func main() {
	m := map[string]int{"a": 1, "b": 2}
	for k, v := range m { // 返回 (key, value)
		println(k, v)
	}

	numbers := []int{1, 2, 3, 4}
	for i, x := range numbers {
		fmt.Printf("第%d 次, x 的值为%d。\n", i, x)
	}
}
```

```
/*
返回:
a 1
b 2
第 0 次, x 的值为 1。
第 1 次, x 的值为 2。
第 2 次, x 的值为 3。
第 3 次, x 的值为 4。
*/
```

返回第一个值为索引值（键值），有时候并不是我们所需要的，因此可以使用“_”空标识符表示忽略第一个返回值。对于空字典或切片、空数组、空字符串等情况，for 语句会直接结束，不会循环。

但如果需要指定 for 执行循环的次数，例如需要获取数组（或者其他支持的类型）里面的值，而数组中有一个值是空字符串，则可以指定数组（或其他支持类型）长度，强制让 for 循环执行相应次数，例如将上面的例子稍微改一下：

```
package main

import "fmt"

func main() {
      numbers := [5]int{1, 2, 3, 4}
      for i, x := range numbers {
            fmt.Printf("第%d 次, x 的值为%d。\n", i, x)
      }
}
/*
返回:
第 0 次, x 的值为 1。
第 1 次, x 的值为 2。
第 2 次, x 的值为 3。
第 3 次, x 的值为 4。
第 4 次, x 的值为 0。
*/
```

由于定义了 numbers 长度为 5，但 numbers 中只有 4 个值，因此最后一个为空值，从 for 循环返回的信息可以看到第 5 次 x 的值为 0，代码块的确执行了 5 次。

4.4 延迟语句

除了上面传统的流程控制语句之外，Go 语言还有一些特殊的控制语句。defer 就是其

中一个，defer 用于延迟调用指定函数，defer 关键字只能出现在函数内部，例如：

```
func main() {
  defer fmt.Println("World")
  fmt.Println("Hello")
}
/*
返回：
Hello
World
*/
```

上面例子会首先打印 Hello，然后再打印 World，因为第一句使用了 defer 关键字，defer 语句会在函数最后执行，被延迟的操作是 defer 后面的内容。

defer 后面的表达式必须是外部函数的调用，上面的例子就是针对 fmt.Println 函数的延迟调用。defer 有两大特点：

（1）只有当 defer 语句全部执行，defer 所在函数才算真正结束执行。

（2）当函数中有 defer 语句时，需要等待所有 defer 语句执行完毕，才会执行 return 语句。

因为 defer 的延迟特点，可以把 defer 语句用于回收资源、清理收尾等工作。使用 defer 语句之后，不用纠结回收代码放在哪里，反正都是最后执行。

这里需要注意 defer 的执行时机，例如下面的例子：

```
package main

import (
      "fmt"
)

var i = 0

func print() {
      fmt.Println(i)
}
func main() {
      for ; i < 5; i++ {
            defer print()
      }

}
/*
```

```
返回:
5
5
5
5
5
*/
```

上面例子返回的是 5 个 5，这是因为每个 defer 都是在函数轮询之后，最后才执行，此时 i 的值当然就是 5 了。如果要正确反向打印数字则应该这样写：

```
package main

import (
      "fmt"
)

var i = 0

func print(i int) {
      fmt.Println(i)
}

func main() {
      for ; i < 5; i++ {
            defer print(i)
      }

}

/*
返回:
4
3
2
1
0
*/
```

上面例子引入了函数参数，虽然还没介绍函数的概念，但是不妨碍理解这里面的 defer 关键知识。这里之所以是一个反序的数字列表，是因为 defer 其实是一个栈，遵循先入后出，或者理解为后进先出。

当 i 等于 0 时，defer 语句第一次被压栈，此时 defer 后面的函数返回 0；i 不断自增，一直到 i 等于 4 时，defer 语句第 5 次入栈，defer 后的函数返回 4；此时 i 的自增不再满足 for 条件，于是跳出循环，在结束之前，Go 语言会根据 defer 后进先出原则逐条打印栈内

的数值，于是就出现现在看到的结果了。

简单来说就是当一个函数内部有多个 defer 语句时，最后面的 defer 语句最先执行（当然是指在所有 defer 语句中）。

4.5　标签

在 Go 语言中，有一个特殊的概念就是标签，可以给 for、switch 或 select 等流程控制代码块打上一个标签，配合标签标识符可以方便跳转到某一个地方继续执行，有助于提高编程效率。标签格式如下：

```
L1:
    for i := 0; i <= 5; i++ {
        // 代码块
    }
/* 下面写法也可以，不过 Go 语言编译器会自动格式化为上面的格式 */
L2:switch i {
        // 代码块
    }
```

标签的名称是区分大小写的，为了提高代码易读性，建议标签名称使用大写字母和数字。标签可以标记任何语句，并不限定于流程控制语句，未使用的标签会引发错误。

4.5.1　break

break 语句意思是打断，这个语句表示打断当前流程控制，例如：

```
package main

import (
      "fmt"
)

func main() {
      for i := 0; i < 10; i++ {
            fmt.Printf("i 的值是：%d\n", i)
            if i > 4 {
                  break
            }
      }

}
/*
```

```
返回:
i 的值是: 0
i 的值是: 1
i 的值是: 2
i 的值是: 3
i 的值是: 4
i 的值是: 5
*/
```

上面例子中 break 的用法与主流编程语言并无差别，当 i > 4 时，break 语句打断了后续的循环操作，跳出循环。

但是对于嵌套流程控制语句，例如下面这种情况：

```
package main

import (
      "fmt"
)

func main() {
      for {
            x := 1
            switch {
            case x > 0:
                  fmt.Println("A")
                  break
            case x == 1:
                  fmt.Println("B")
            default:
                  fmt.Println("C")
            }
      }
}
/*
返回:
A
... ...
*/
```

上面的例子会一直返回 A，无限循环。上面代码出现的 break 语句只是跳出了 switch 流程，并没有跳出 for 循环，所以这个程序会一直执行下去。

为了跳出指定的流程控制代码块，就需要标签出场了：

```
package main

import (
```

```
    "fmt"
)

func main() {
LOOP1:
    for {
        x := 1
        switch {
        case x > 0:
            fmt.Println("A")
            break LOOP1
        case x == 1:
            fmt.Println("B")
        default:
            fmt.Println("C")
        }
    }
}
/*
返回:
A
*/
```

上面代码中的 break LOOP1 表示跳出到标签为 LOOP1 的代码块之外。

4.5.2　continue

与 break 相反，continue 用于跳转到指定代码块位置继续执行任务，continue 仅能用于 for 循环。例如与上面几乎一样的代码，只是改动了一个关键字：

```
package main

import (
    "fmt"
)

func main() {
LOOP1:
    for i := 0; i <= 5; i++ {
        switch {
        case i > 0:
            fmt.Println("A")
            continue LOOP1
        case i == 1:
            fmt.Println("B")
        default:
            fmt.Println("C")
```

```
            }
            fmt.Printf("i is: %d\n", i)
      }
}

/*
返回:
C
i is: 0
A
A
A
A
A
*/
```

上面代码中因为使用了 continue 语句跳转到了外围 for 循环中，与 break 不同的是，continue 表示跳转后继续执行操作。

4.5.3 goto

Go 语言中的 goto 语句可以无条件跳转到相同函数中的带标签语句。标签、goto 等关键字都并非 Go 语言独创，Go 语言可以说是一门大量参考了其他语言优点的编程语言，在流程控制上做了一些扩增（如类型 switch），同时也减少了一些关键字（如 while）。

```
package main

func main() {
      var i int
      for {
            println(i)
            i++
            if i > 2 {
                  goto BREAK
            }
      }
BREAK:
      println("break")
}

/*
返回:
0
1
2
```

```
break
*/
```

上面代码中，goto 语句指向了 BREAK 标签，所以当循环中遇到 i>2 时，直接跳转打印 break。goto 只能在同一个函数中跳转。

4.6　本章小结

在这一章中介绍了 4 种 Go 语言流程控制语句，通过本章的学习，我们已经可以编写简单的 Go 语言程序了，流程控制语句是一切代码的基础，合理使用流程控制可以使程序执行效率更高。下一章将开始函数的学习。

第5章 函数

在前4章所介绍的内容中，已经详细讲解了Go语言顺序编程（过程式编程）中的所有基本语法概念，本章将在此基础上介绍函数的概念。函数是包含用于执行某一任务或计算的一系列语句，通过对一系列语句的包装，使其更便于阅读，也使其更加简明、清晰、便捷。

我们已经认识了main函数，知道它是程序的入口，而且因为Go语言是编译型语言，所以函数编写的顺序是无关紧要的，但为了可读性，最好把main()函数写在文件的前面，其他函数按照一定逻辑顺序进行编写（函数被调用的顺序、函数的用途等）。

5.1 认识函数

对于熟悉Java或者其他编程语言的读者来说，对于函数和方法可能有些误解，在Go语言中，函数和方法不太一样，有明确的区分。但对于Java而言，一般可以认为函数就是方法，方法就是函数。在Go语言中，函数是指不属于任何结构体、类型的方法，也就是说，函数是没有接收者的；而方法是有接收者的，我们说的方法要么属于一个结构体，要么属于一个新定义的类型。

5.1.1 函数的声明

函数的定义声明前面已经出现过，下面例子中定义了add函数，它的函数声明是func add(a, b int) int，直接定义在一个包之下，可以直接调用，比如例子中的main函数就直接调用了add函数。

```
func main() {
    sum := add(1, 2)
    fmt.Println(sum)
}

func add(a, b int) int {
    return a + b
}
```

一个完整的函数声明结构格式如下：

```
func funcName(input1 type1, input2 type2) (output1 type1, output2 type2) {
    // 逻辑代码
    return value1, value2  // 返回多值
}
```

从以上代码可以看出关键字 func 是用来声明函数的，而 funcName 是指函数名称（匿名函数和 lambda 函数除外）。函数名称如果是小写开头的，它的作用域只属于所声明的包，不能被其他包调用；如果函数名以大写字母开头，则该函数便是公开的，可以被其他包调用。这也是 Go 语言中大小写的用处，这个规则适用于所有变量、函数等实体对象的声明，类似 Java 中的作用域关键字（private、protect、public 等）。

注意，Go 语言函数不支持嵌套（nested）、重载（overload）和默认参数（default parameter）。

声明一个在外部定义的函数只需给出函数名与函数签名，不需要写出完整的函数体：

```
func hello(str, num int) // 外部实现
```

函数同样可以通过声明的方式作为一个函数类型被使用，就像下面这样：

```
type addNum func(int, int) int
```

此处不需要函数体{}，因为函数在 Go 语言中属于一等值（first-class value），函数也可以赋值给变量，例如 add := addNum，由于变量指向了 addNum 函数的签名，所以不能再给它赋一个具有不同签名的函数值。

不过函数值（functions value）之间倒是可以相互比较的，例如它们引用的是相同的函数或者返回值都是 nil，则可以认为它们是相同的函数。函数不能在其他函数里面声明，也就是不能嵌套（匿名函数除外）。

5.1.2 函数的参数

函数可以有一个或者多个参数，每个参数后面带有类型，通过“,”符号分隔。如果

参数列表中若干个相邻参数的类型相同，比如上面例子中的 a 和 b，则可以在参数列表中省略前面变量的类型声明，如下所示：

```
func Add(a, b int)(ret int, err error) {
    // ...
}
```

如果返回值列表中多个返回值的类型相同，也可以用同样的方式合并。如果函数只有一个返回值，也可以这样写：

```
func Add(a, b int) int {
    // ...
}
```

从其他编程语言转过来的读者，可能更习惯这种写法。

5.1.3 函数的返回值

返回值就是函数执行后返回的内容，函数可以返回多个值，前一节的示例代码声明了两个返回值变量 output1 和 output2。对于多返回值函数，如果不声明变量，也需要指定返回类型。

如果只有一个返回值且不声明返回值变量，那么可以省略返回值（包括返回值的括号都可以不写）。如果没有返回值，那么就直接省略最后的返回信息，什么都不用写；如果有返回值，那么必须在函数的最后添加 return 语句。

例如以下程序定义了一个函数 max，用于判断两个数字之间的大小：

```
// 示例代码 5-1
package main

import "fmt"

func main() {
    x := 1
    y := 2
    z := 3
    var maxNum int
    maxNum = max(x, y)
    fmt.Printf("max(%d, %d) = %d\n", x, y, maxNum)
    maxNum = max(x, z)
    fmt.Printf("max(%d, %d) = %d\n", x, z, maxNum)
    fmt.Printf("max(%d, %d) = %d\n", y, z, max(y, z)) // 直接调用
}
```

```
// max 私有函数
func max(a, b int) (maxNum int) {
      if a > b {
            return a
      }
      return b
}

// Max 共有函数
func Max(a, b int) (maxNum int) {
      if a > b {
            maxNum = a
      }
      maxNum = b
      return maxNum
}
/*
返回:
max(1, 2) = 2
max(1, 3) = 3
max(2, 3) = 3
*/
```

在以上例子中可以看到函数 max 的声明：func max(a, b int) int，其中 a 和 b 都是 int 类型，所以可以合并写在一起（等同于 a int, b int），而返回值只有一个，所以可以只写返回值的类型，也可以在声明时指定返回值的变量名，例如：func max(a, b int) m int，对于多返回值的情况将在下一节介绍。在函数块里，return 之后的语句都不会执行。

在上面的代码中，出现了两个功能相同的函数，其中一个为 max，另一个为 Max，这两个函数的区别在于外部是否能够调用。

此外，有些函数只是完成一个任务，并没有返回值。我们仅仅是利用了这种函数的副作用，就像输出文本到终端、发送一个邮件或者记录一个错误等。但是绝大部分函数还是有返回值的。

5.2 函数的基础

Go 语言函数有很多比较先进的特性，例如多返回值、延迟语句、可变参数等。掌握这些函数特点可以更高效地编写 Go 语言的函数。

5.2.1 多返回值

相比 C、C++、Java 和 C#，多返回值是 Go 语言的一大特性。函数能够接收参数供自己使用，也可以返回一组值，这为判断一个函数是否正常执行提供了方便。直接看下面的例子：

```
// 示例代码 5-2
package main

import "fmt"

// SumAndProduct 返回 A+B 和 A*B
func SumAndProduct(A, B int) (int, int) {
      return A + B, A * B
}

func main() {
      x := 3
      y := 4
      xPLUSy, xTIMESy := SumAndProduct(x, y)
      fmt.Printf("%d + %d = %d\n", x, y, xPLUSy)
      fmt.Printf("%d * %d = %d\n", x, y, xTIMESy)
}
/*
返回:
3 + 4 = 7
3 * 4 = 12
*/
```

从上面的例子可以看到直接返回了两个参数，这里也可以命名返回参数的变量，但在上面例子中只使用 int 类型注明返回两个变量（返回不注明变量名，是因为直接在函数里面初始化了）。也可以改成下面这样定义：

```
func SumAndProduct(A, B int) (add int, Multiplied int) {
      add = A+B
      Multiplied = A*B
      return
}
```

如果你的函数是导出的（首字母大写），最好命名返回值，因为不命名返回值，虽然使得代码更加简洁了，但是会造成生成的文档可读性差。

多返回值也意味着有时候返回值并不是全部都需要，还记得之前使用过的空白标识符“_”吧，它同样可以用于忽略函数返回值。例如下面的 ThreeValues 是拥有三个返回值的函数。

```
// 示例代码5-3
package main

import "fmt"

func main() {
    var i1 int
    var f1 float32
    i1, _, f1 = ThreeValues()
    fmt.Printf("The int: %d, the float: %f \n", i1, f1)
}

func ThreeValues() (int, int, float32) {
    return 5, 6, 7.5
}
```

代码中将第一个与第三个返回值赋给了i1与f1；第二个返回值赋给了空白标识符“_”，然后自动丢弃掉。

5.2.2 函数作为参数

函数是第一类对象，可作为参数传递。只要被调用函数的返回值个数、返回值类型和返回值的顺序与调用函数所需求的实参是一致的，就可以把这个被调用的函数当作其他函数的参数。

假设f1需要3个参数，同时f2返回3个参数，就可以像下面这样调用f1：

```
func f1(a, b, c int)
func f2(a, b int) (int, int, int)
func f1(f2(a, b))
```

建议将复杂签名定义为函数类型，以便于阅读。看下面的例子：

```
// 示例代码5-4
package main

import "fmt"

func pipe(ff func() int) int {
      return ff()
}

// FormatFunc 定义函数类型
type FormatFunc func(s string, x, y int) string

func format(ff FormatFunc, s string, x, y int) string {
      return ff(s, x, y)
```

```
}

func main() {
	s1 := pipe(func() int { return 100 }) // 直接将匿名函数当参数
	s2 := format(func(s string, x, y int) string {
		return fmt.Sprintf(s, x, y)
	}, "%d, %d", 10, 20)
	fmt.Println(s1, s2)
}
/*
返回:
100 10, 20
*/
```

有返回值的函数，必须有明确的终止语句（return 或 panic），否则会引发编译错误。

5.2.3 函数作为类型

函数在 Go 语言中也是一种变量，前面多次通过 type 来定义它，它的类型就是所有拥有相同参数与相同返回值的一种函数类型。

```
type typeName func(input1 inputType1 , input2 inputType2 [, ...]) (result1 resultTy
pe1 [, ...])
```

函数作为类型的最大好处在于可以把这个类型的函数当作值来传递，请看下面的例子：

```
// 示例代码 5-5
package main

import "fmt"

func isOdd(v int) bool {
	if v%2 == 0 {
		return false
	}
	return true
}

func isEven(v int) bool {
	if v%2 == 0 {
		return true
	}
	return false
}

type boolFunc func(int) bool // 声明一个函数类型

// 函数作为一个参数使用
```

```
func filter(slice []int, f boolFunc) []int {
	var result []int
	for _, value := range slice {
		if f(value) {
			result = append(result, value)
		}
	}
	return result
}

func main() {
	slice := []int{3, 1, 4, 5, 9, 2}
	fmt.Println("slice = ", slice)
	odd := filter(slice, isOdd) // 函数当作值来传递
	fmt.Println("odd:", odd)
	even := filter(slice, isEven) // 函数当作值来传递
	fmt.Println("even:", even)
}

/*
返回:
slice =  [3 1 4 5 9 2]
odd: [3 1 5 9]
even: [4 2]
*/
```

在写一些通用接口的时候，把函数当作值和类型会非常有用，上面例子中的 boolFunc 就是一个函数类型，两个 is*函数的参数和返回值与 boolFunc 类型一样，通过这种模式可以实现各种各样逻辑，使得程序变得非常灵活。

5.2.4 可变参数

函数有着不定数量的参数在很多时候都会遇到。Go 语言函数支持可变参数（简称变参），为了做到这一点，首先需要定义函数使其接收变参：

```
func myfunc(arg ...int) {}
```

在 Go 语言中，函数的最后一个参数如果是...type 的形式，那这个函数就是一个变参函数，它可以处理变长的参数，而这个长度可以是 0。注意的是在变参函数中，无论变参有多少个，它们的类型全部都一样。以上面的函数为例，函数体中变量 arg 是一个 int 类型的切片（slice），所以这个函数的参数是一个类似 int 类型的切片，该参数可以通过前面提到的 for 循环结构来迭代，看下面代码：

```
for _, n := range arg {
    fmt.Printf("The number is: %d\n", n)
}
```

变参函数的“变参”可以是函数的其中一个参数，例如下面的代码获取一堆数字中的最大或最小值，假设这堆数字被存储在数组 arr 中，则可以通过 arr...的形式来传递参数调用变参函数。

```
// 示例代码 5-6
package main

import "fmt"

func main() {
    // 手动填写参数
    age := ageMinOrMax("min", 1, 3, 2, 0)
    fmt.Printf("年龄最小的是%d 岁。\n", age)
    // 数组作为参数
    ageArr := []int{7, 9, 3, 5, 1}
    age = ageMinOrMax("max", ageArr...)
    fmt.Printf("年龄最大的是%d 岁。", age)
}

func ageMinOrMax(m string, a ...int) int {
    if len(a) == 0 {
        return 0
    }
    if m == "max" {
        max := a[0]
        for _, v := range a {
            if v > max {
                max = v
            }
        }
        return max
    } else if m == "min" {
        min := a[0]
        for _, v := range a {
            if v < min {
                min = v
            }
        }
        return min
    } else {
        e := -1
        return e
    }
}
```

```
/*
返回:
年龄最小的是0岁。
年龄最大的是9岁。
*/
```

以上函数同时使用了两种不同类型的参数，不同类型的参数要单独声明，而可变参数必须是同一种类型（就目前而言）。

下面这个例子中的函数都是接收一个变参并打印每个元素，比较函数 f2 和 f3 的区别。一个接收变参的函数可以将这个变参作为其他函数的参数进行传递。

```
// 示例代码5-7
package main

import "fmt"

func main() {
	ageArr := []int{7, 9, 3, 5, 1}
	f1(ageArr...)
}

func f1(arr ...int) {
	f2(arr...)
	fmt.Println("")
	f3(arr)
}

func f2(arr ...int) {
	for _, char := range arr {
		fmt.Printf("%d ", char)
	}
}
func f3(arr []int) {
	for _, char := range arr {
		fmt.Printf("%d ", char)
	}
}
/*
返回:
7 9 3 5 1
7 9 3 5 1
*/
```

变长参数可以作为对应类型的slice进行二次传递。从内部实现机制上来说，类型...type本质上是一个数组切片，也就是[]type，这也是 arr 可以用 for 循环来获得每个元素的值的原因。

函数 f2 和 f3 的最大区别在于调用方式：

```
f2(1, 3, 7, 13) // 可变参数，随便写
f3([]int{1, 3, 7, 13}) // 需要加上[]int{}来构造一个数组切片实例
```

最后是关于可变参数为多种类型的情况，之前的例子都是把变参类型约束为一种类型，但如果希望传递任意类型，可以指定类型为 interface{}，也就是接口。下面是 Go 语言标准库中 fmt.Printf()的函数原型：

```
func Printf(format string, args ...interface{}) {
    // ...
}
```

常用的 Printf 函数就是一个变参函数，而且不限制参数类型。用 interface{}传递任意类型数据是 Go 语言的习惯用法。interface{}是类型安全的，这和 C/C++不太一样，关于它的用法在第 9 章会介绍。除了使用空接口解决这个问题，还可以使用结构体解决，关于这部分内容会在第 8 章介绍。

5.2.5 匿名函数与闭包

1. 匿名函数

匿名函数是指不需要定义函数名的一种函数定义方式，匿名函数在很多语言中都有，比如 JavaScript、C#和 Java 等语言都提供了匿名函数。如下是一个简单的匿名函数的例子：

```
func(x, y int) int { return x + y }
```

在这个函数声明中并没有命名函数名称，所以这样的一个函数不能够独立存在，但可以被赋值于某个变量，即保存函数的地址到变量中：

```
fplus := func(x, y int) int { return x + y }
fplus(3,4)
```

通过变量名对函数进行调用，这个例子返回数字 7。除此之外还可以直接调用匿名函数：

```
func(x, y int) int { return x + y } (3, 4)
```

后面紧跟函数运行参数，同样返回数字 7。一个完整的例子如下：

```
// 示例代码 5-8
package main

import (
    "fmt"
```

```
)

func main() {
	func(num int) int {
		sum := 0
		for i := 1; i <= num; i++ {
			sum += i
		}
		fmt.Println(sum)
		return sum
	}(100)
}
/*
返回:
5050
*/
```

参数列表的第一对括号必须紧挨着关键字 func，因为匿名函数没有名称。花括号{}涵盖着函数体，最后的一对括号表示对该匿名函数的调用。

2. 闭包

匿名函数同样被称为闭包（函数式语言的术语），简单来说闭包允许调用定义在其他环境下的变量，可使得某个函数捕捉到一些外部状态，例如函数被创建时的状态。用专业的语言表述就是：一个闭包继承了函数声明时的作用域。这种状态（作用域内的变量）会共享到闭包的环境中，因此这些变量可以在闭包中被操作，直到被销毁。闭包经常被用作包装函数，预先定义好一个或多个参数以用于包装，另一种常见的应用就是使用闭包来完成更加简洁的错误检查。

```
// 示例代码 5-9
package main

import "fmt"

func main() {
	fmt.Printf("%d\n", Add()(3))
	fmt.Printf("%d\n", Add2(6)(3))
}

// Add 无参函数，返回值是一个匿名函数，匿名函数返回一个 int 类型的值
func Add() func(b int) int {
	return func(b int) int {
		return b + 2
	}
}
```

```
// Add2 无参函数，返回值是一个匿名函数，匿名函数返回一个 int 类型的值
func Add2(a int) func(b int) int {
      return func(b int) int {
            return a + b
      }
}
/*
返回:
5
9
*/
```

在以上代码中，函数 Add 不接收任何参数，但函数 Add2 接收一个 int 类型的整数作为参数。

下面来看一个简单的例子，以便理解闭包继承函数声明时的作用域这个特点：

```
// 示例代码 5-10
package main

import (
      "fmt"
)

func main() {
      j := 5
      a := func() func() {
            i := 10
            return func() {
                  fmt.Printf("i = %d j = %d\n", i, j)
            }
      }()

      a()
      j = 10
      a()
}
/*
返回:
i = 10 j = 5
i = 10 j = 10
*/
```

在上面这个例子中，可以看到 j 变量在闭包之外初始化，当闭包第一次获取 j 的值时返回的是初始化的值，但第二次获取时，闭包返回了赋值之后的 j 的值。在变量 a 指向的闭包函数中，只有内部的匿名函数才能访问变量 i，而无法通过其他途径访问到，因此保

仅供非商业用途或交流学习使用

证了 i 的安全性。这很好理解，现在来看一个稍微不一样的写法。

```
// 示例代码 5-11
package main

import "fmt"

func main() {
      var f = adder()
      // 相当于 fmt.Println(adder()(1))
      fmt.Println(f(1))
      fmt.Println(f(2))
      fmt.Println(f(3))
}

func adder() func(int) int {
      var x int // 闭包中的变量可以在闭包函数体内声明，也可以在外部函数声明
      return func(d int) int {
            fmt.Println("x = ", x, "d = ", d)
            x += d
            return x
      }
}
/*
返回：
x =  0 d =  1
1
x =  1 d =  2
3
x =  3 d =  3
6
*/
```

三次调用函数 adder()中变量 d 的值分别为：1、2 和 3。在多次调用中，变量 x 的值是被保留的，即 0+1=1，然后 1+2=3，最后 3+3=6，闭包函数保存并积累其中的变量的值，不管外部函数是否退出，它都能够继续操作外部函数中的局部变量，这些局部变量同样可以是参数。这样说可能不是很好理解，所以再换一种直观的写法：

```
// 示例代码 5-12
package main

import "fmt"

func getSequence() func() int {
      i := 0
      return func() int {
            i++
```

仅供非商业用途或交流学习使用


```
            return i
        }
    }

    func main() {
        // nextNumber 为一个函数，函数 i 为 0
        nextNumber := getSequence()

        // 调用 nextNumber 函数，i 变量自增 1 并返回
        fmt.Printf("%d ", nextNumber())
        fmt.Printf("%d ", nextNumber())
        fmt.Printf("%d ", nextNumber())

        // 创建新的函数 nextNumber1，并查看结果
        nextNumber1 := getSequence()
        fmt.Printf("%d ", nextNumber1())
        fmt.Printf("%d ", nextNumber1())
    }
    /*
    返回：
    1 2 3 1 2
    */
```

这个例子清晰地展示了 Go 语言闭包在环境继承方面的特性。

5.2.6 递归函数

递归，就是在运行的过程中调用自己。语法格式如下：

```
func recursion() {
   recursion() /* 函数调用自身 */
}

func main() {
   recursion()
}
```

Go 语言支持递归，但在使用递归时，开发者需要设置退出条件，否则递归将陷入无限循环。递归函数对于解决数学上的问题是非常有用的，就像计算阶乘、生成斐波那契数列等。以下实例通过 Go 语言的递归函数实现阶乘运算。

```
// 示例代码 5-13
package main

import "fmt"

func Factorial(n uint64)(result uint64) {
```

仅供非商业用途或交流学习使用


```
    if (n > 0) {
        result = n * Factorial(n-1)
        return result
    }
    return 1
}

func main() {
    var i int = 15
    fmt.Printf("%d 的阶乘是 %d\n", i, Factorial(uint64(i)))
}
/*
返回:
15 的阶乘是 1307674368000
*/
```

以下实例通过 Go 语言的递归函数实现斐波那契数列：

```
// 示例代码 5-14
package main

import "fmt"

func fibonacci(n int) int {
  if n < 2 {
   return n
  }
  return fibonacci(n-2) + fibonacci(n-1)
}

func main() {
    var i int
    for i = 0; i < 10; i++ {
       fmt.Printf("%d\t", fibonacci(i))
    }
}
/*
返回:
0    1    1    2    3    5    8    13    21    34
*/
```

递归可以优雅地解决很多问题，例如著名的快速排序算法就是通过递归实现的。递归函数在使用时有可能遇到栈溢出的问题，当大量的递归调用导致的程序栈内存分配耗尽就会产生栈溢出。好在通过懒惰求值的办法可以解决这个问题，在 Go 语言中，可以通过管道（channel）和 goroutine 来解决这个问题，后面会介绍。

仅供非商业用途或交流学习使用

5.2.7 内置函数

内置函数就是不需要进行导入操作就可以直接使用的函数，它们有些可以根据不同的类型进行操作（例如 len、cap 和 append），有些用于系统级的操作（例如 panic）。不管哪种，它们都需要直接获得编译器的支持。

常见内置函数如下所示。

- close：用于管道通信。
- len：用于返回某个类型的长度或数量（字符串、数组、切片、map 和管道）。
- cap：容量的意思，用于返回某个类型的最大容量（只能用于切片和 map）。
- new、make：均用于分配内存，不过 new 用于值类型和用户定义的类型，如自定义结构，make 用于内置引用类型（切片、map 和管道）。它们的用法就像是函数，但是将类型作为参数：new(type)、make(type)。new(T)分配类型 T 的零值并返回其地址，也就是指向类型 T 的指针，它也可以用于基本类型：v := new(int)。make(T)返回类型 T 的初始化之后的值，因此它比 new 做更多的工作。new()是一个函数，不要忘记它的括号。
- copy、append：用于复制和连接切片。
- panic、recover：两者均用于错误处理机制。
- print、println：底层打印函数（部署环境中建议使用 fmt 包）。
- complex、real imag：用于创建和操作复数。

在后面的章节中如果遇到这些内置函数再详细介绍。

5.3 函数进阶

5.3.1 参数传递机制

Go 语言的参数传递可以分为“按值传递”和“按引用传递”。Go 语言默认按值传递，传递的是参数的副本，函数接收参数副本后，使用变量的过程中可能对副本的值进行更改，但不会影响原来的变量。换句话说，调用函数时修改参数的值，不会影响原来的实参的值，因为数值变化只作用在副本上。

所以如果要让函数直接修改参数的值，而不是对参数的副本进行修改，就需要将参数的地址（变量名前面添加&符号，比如&variable）传递给函数，这就是按“引用传递”，

仅供非商业用途或交流学习使用

此时传递给函数的是一个指针。

如果把指针传递给函数，指针的值（一个地址）就会被复制，但指针的值指向的地址上的那个值不会被复制（被复制的是指针，但是两个指针指向同一个实际的值）。这样一来，修改这个指针的值，实际上意味着这个值所指向的地址上的值被修改了（别忘了指针也是变量类型，有自己的地址和值，通常指针的值指向一个变量的地址，所以说到底按引用传递其实也是按值传递）。

为了验证上面的说法，我们来看一个例子：

```
// 示例代码 5-15
package main

import "fmt"

func main() {
      x := 3
      fmt.Println("x =", x, "&x =", &x)

      y := add(x)                       // 执行 add 函数，但实际上修改的是 x 的副本
      fmt.Println("x =", x, "y =", y) // 输出的 x 还是原来的值，改变的是副本 y 的值

      z := addP(&x)                     // 执行 addP 函数，实际上修改的是 x 的值
      fmt.Println("x =", x, "z =", z) // 输出的 x 已经被修改
      fmt.Println("&x =", &x)         // x 地址还是原来的
}
func add(a int) int {
      a++
      return a
}
func addP(a *int) int {
      *a++
      return *a
}
/*
返回：
x = 3 &x = 0xc4200160c8
x = 3 y = 4
x = 4 z = 4
&x = 0xc4200160c8
*/
```

add()函数中 x 变量的值没有发生变化，而 addP()函数中 x 的值变了。当调用 add()的时候，add()接收的参数其实是 x 的副本，而不是 x 本身；而 addP()函数接收的参数是一个指针，指向的是 x 的本身，所以 x 实际的值就改变了。

仅供非商业用途或交流学习使用

传指针最明显的三点好处如下。

第一，传指针使得多个函数能操作同一个对象。

第二，传指针比较轻量级（8B），毕竟只是传内存地址，可以用指针传递体积大的结构体。如果传递值，在每次创建副本上面就会花费相对较多的系统开销（内存和时间）。所以当要传递大的结构体的时候，用指针是一个明智的选择。一般来说，传递指针（一个32 位或者 64 位的值）的消耗比传递副本占用更少资源。在函数调用时，像切片（slice）、字典（map）、接口（interface）、通道（channel）这样的引用类型都是默认使用引用传递的（即使没有显式地指出指针）。

不过需要注意的是，Go 语言中 channel、slice、map 这三种类型的实现机制类似指针，所以可以直接传递，而不用取地址后传递指针。不过若函数需改变 slice 的长度，则仍需要取地址传递指针。

第三，传递指针给函数不但可以节省内存（因为没有复制变量的值），而且赋予了函数直接修改外部变量的能力，所以被修改的变量不再需要使用 return 返回。

关于第三点，来看一个例子，下面代码 res 是一个指向 int 变量的指针，通过这个指针，在函数内修改了这个 int 变量的数值：

```
// 示例代码 5-16
package main

import (
      "fmt"
)

func main() {
      n := 0
      res := &n
      multiply(2, 4, res)
      fmt.Println("Result:", *res)
}

func multiply(a, b int, res *int) {
      *res = a * b
}
/*
返回：
Result: 8
*/
```

上面仅仅是个例子，实际开发中传递一个指针很容易引发一些不确定的事，请万分小

仅供非商业用途或交流学习使用

心那些可以改变外部变量的函数，迫不得已使用时请添加注释以便他人能够清楚这个函数到底做了什么。

尽管如此，传递指针的优势还是十分明显的，当需要在函数内改变一个占用内存比较大的变量时，传递指针可以极大减少内存占用，性能优势尤为出色，所以这把双刃剑要适当利用。

5.3.2　defer 与跟踪

1．defer 延迟语句

Go 语言中让人颇为称道的一个设计就是延迟（defer）语句，开发者可以在函数中添加多个 defer 语句。当函数执行到最后时（return 语句执行之前），这些 defer 语句会按照“逆序”执行，最后该函数才退出。

1）defer 的用途

在进行一些 IO 操作的时候，如果遇到错误，便需要提前返回，而返回之前应该关闭相应的资源，否则容易造成资源泄露等问题，defer 语句就可以出色地解决这个问题。对于其他语言而言，打开一个资源一般这样写：

```
func ReadWrite() bool {
      file.Open("file")
      // 做些什么之后
      if aFailure { // 遇到状况 a，需要提前返回
            file.Close()
            return false
      } else if bFailure { // 遇到状况 b，需要提前返回
            file.Close()
            return false
      }
      // 正常执行，关闭资源
      file.Close()
      return true
}
```

上面的例子中有很多重复的代码，而使用 Go 语言的 defer 语句可以解决这个问题。下面看看 Go 语言的写法，使用 defer 不但使得代码量减少了很多，而且程序变得更优雅简洁：

```
func ReadWrite() bool {
      file.Open("file")
      defer file.Close() // 打开和关闭写在一起方便管理，还不容易遗漏
```

仅供非商业用途或交流学习使用


```
        if aFailure {
                return false
        } else if bFailure {
                return false
        }
        return true
}
```

在 defer 后指定的函数会在函数退出前调用。如果多次调用 defer，那么 defer 采用后进先出模式，例如以下代码：

```
for i := 0; i < 5; i++ {
        defer fmt.Printf("%d ", i)
}
/*
返回:
4 3 2 1 0
*/
```

2）执行顺序

关于 defer、return 和返回值之间的执行顺序，很多 Go 语言新手都会掉进坑里，下面用几个例子说明。

首先需要特别注意函数的返回值是否被命名，例如下面是无名返回值的情况：

```
// 示例代码 5-17
package main

import (
        "fmt"
)

func main() {
        fmt.Println("return:", a()) // 打印结果为 return: 0
}

func a() int {
        var i int
        defer func() {
                i++
                fmt.Println("defer2:", i) // 打印结果为 defer2: 2
        }()
        defer func() {
                i++
                fmt.Println("defer1:", i) // 打印结果为 defer1: 1
        }()
        return i
}
```

仅供非商业用途或交流学习使用


```
/*
返回:
defer1: 1
defer2: 2
return: 0
*/
```

以下为有名返回值的情况：

```
// 示例代码5-18
package main

import (
	"fmt"
)

func main() {
	fmt.Println("return:", b()) // 打印结果为 return: 2
}

func b() (i int) {
	defer func() {
		i++
		fmt.Println("defer2:", i) // 打印结果为 defer2: 2
	}()
	defer func() {
		i++
		fmt.Println("defer1:", i) // 打印结果为 defer1: 1
	}()
	return i // 或者直接 return 效果相同
}
/*
返回:
defer1: 1
defer2: 2
return: 2
*/
```

首先我们知道，多个 defer 语句的执行顺序为“逆序”，defer、return、返回值三者的执行逻辑应该是：defer 最先执行一些收尾工作；然后 return 执行，return 负责将结果写入返回值中；最后函数携带当前返回值退出。

上面两段代码的返回结果之所以不同，是因为 a() int {}的返回值没有被命名，而且需要特别注意的是，尽管两个 defer 语句的匿名函数也没有返回值，但是匿名函数的特点是可以继承变量的值，所以 defer2 语句继承了 defer1 语句的值。按理说 a()的返回值是 2 才对，然而返回值却是 0，这是因为返回值没有被声明，所以函数 a()的返回值还是 0。

仅供非商业用途或交流学习使用

然后再来看 b() (i int) {}的返回值，已经声明为 i，也就意味着 defer 中是可以调用到真实返回值的，因此 defer 在 return 赋值返回值 i 之后，再一次修改了 i 的值，最终函数退出后的返回值才会是 defer 修改过的值。

3）defer 原理

虽然已经了解 defer 与 return 和返回值的执行关系，但是有一点还要深究的是 defer 到底是如何延时调用的，在 defer 中的表达式必须是函数调用，意味着 defer 后面不能执行操作语句。

首先 return 的实现逻辑简单来说就是：

- 第一步是给返回值赋值（若为有名返回值则直接赋值，若为匿名返回值则先声明再赋值）；
- 第二步调用 RET 返回指令并传入返回值，而 RET 则会检查 defer 是否存在，若存在就先逆序插播 defer 语句；
- 最后 RET 携带返回值退出函数。

可以看出，return 并不是一个原子操作，函数返回值与 return 返回值并不一定一致。因此，defer、return、返回值三者的执行顺序应该是：return 最先给返回值赋值；接着 defer 开始执行一些收尾工作；最后 RET 指令携带返回值退出函数。

最后需要特别提醒的是，defer 声明时会先计算确定参数的值，defer 推迟执行的仅是其函数体，因此 defer 语句位置并非随意，defer 的初始化还是受到外部影响的。

```
// 示例代码 5-19
package main

import (
      "fmt"
      "time"
)

func main() {
      i, p := a()
      fmt.Println("return:", i, p, time.Now())
}
func a() (i int, p *int) {
      // i++
      /* 下面是有参数的 defer 函数，函数中的 i 与外围函数的 i 并不是同一个参数，初始化时 i 值
为 0，打印结果为 0 */
      defer func(i int) {
```

仅供非商业用途或交流学习使用


```
        fmt.Println("defer3:", i, &i, time.Now())
    }(i)
    // 下面函数有参数，初始化时 i 的值还是 0，所以延迟打印的结果是 0
    defer fmt.Println("defer2:", i, &i, time.Now())
    // 下面函数无参数，没有参数初始化，i 的值为 return 语句的返回值
    defer func() {
        fmt.Println("defer1:", i, &i, time.Now())
    }()

    i++
    func() {
        fmt.Println("print1:", i, &i, time.Now())
    }()
    fmt.Println("print2:", i, &i, time.Now())
    return i, &i
}
/*
返回：
print1: 1 0xc4200160c8 .... m=+0.000206777
print2: 1 0xc4200160c8 .... m=+0.000308892
defer1: 1 0xc4200160c8 .... m=+0.000316124
defer2: 0 0xc4200160c8 .... m=+0.000205528 // 这个才是最早被初始化的，只不过被延迟执行而已
defer3: 0 0xc4200161b8 .... m=+0.000324588 /* 此处 defer 函数与整个程序没什么关联，i 的
                                              地址也不一致 */
return: 1 0xc4200160c8 .... m=+0.000328836
*/
```

尝试把 i++的位置改为注释的位置，再查看返回结果。

2. 跟踪

defer 还经常用于代码执行追踪，这个方法十分实用，具体为在进入和离开某个函数时打印相关的消息，例如下面两个函数：

```
func trace(s string) string {
    fmt.Println("开始执行", s)
    return s
}
func un(s string) {
    fmt.Println("结束执行", s)
}
```

有了上面两个函数，接下来对任意代码我们都可以调用这两个函数来追踪代码执行，下面例子展示的是 defer 语句的初始化特点：

```
// 示例代码 5-20
package main
```

仅供非商业用途或交流学习使用


```
import "fmt"

func main() {
      b()
}
func a() {
      defer un(trace("a")) // 初始化 defer 函数的参数，所以输出 trace()的结果
      fmt.Println("a 的逻辑代码")
}
func b() {
      defer un(trace("b"))
      fmt.Println("b 的逻辑代码")
      a()
}

func trace(s string) string {
      fmt.Println("开始执行", s)
      return s
}
func un(s string) {
      fmt.Println("结束执行", s)
}
/*
返回:
开始执行 b
b 的逻辑代码
开始执行 a
a 的逻辑代码
结束执行 a
结束执行 b
*/
```

也可以使用 defer 语句来记录函数的参数与返回值，例如：

```
      defer func() {
            log.Printf("func1(%q) = %d, %v", s, n, err)
      }()
```

使用这种办法可以获取函数调用时的输入值和输出值。使用延迟语句还可以修改 return 语句返回的值，比如修改 return 的错误信息等。

5.3.3 错误与恢复

错误处理是现代化编程语言必须慎重考虑的一个话题。几种主流的语言如 C++、Java 等都引入了错误处理的概念，比如异常（exception）的概念和 try-catch 关键字等。Go 语言在错误处理上考虑得更为深远，漂亮的错误处理规范是 Go 语言最大的亮点之一。

仅供非商业用途或交流学习使用

因为Go语言并没有像Java那样提供异常机制，所以Go语言不能抛出异常，取而代之的是提供了panic和recover机制。在展开之前，希望读者记住，应当把panic作为最后的手段来使用，也就是说你的代码中应当没有或者很少有panic语句。panic是个强大的工具，但需要适当地使用它。下面是panic()和recover()函数的声明格式：

```
func panic(interface{})
func recover() interface {}
```

在开始介绍Go语言的panic等机制之前，先来认识Go语言是怎样面对程序错误的。

1. error

Go语言通过error接口实现错误处理的标准模式，该接口的定义如下：

```
type error interface {
    Error() string
}
```

对于大部分函数而言，返回错误基本上都可以定义为以下模式，即将error作为多返回值中的最后一个返回，当然这并非强制要求。

```
func Foo(param int) (n int, err error) {
    // ...
}
```

调用时，代码一般会按下面这种方式处理错误情况：

```
func main() {
    n, err := Foo(0)
    if err != nil {
        // 错误处理
    } else {
        // 使用返回值 n
    }
}
```

这里使用Go语言标准库中的实际代码来演示如何使用自定义的error类型。首先需要定义一个类型，用于承载错误信息，不过因为Go语言中接口十分灵活，所以不需要从error接口继承或者像Java一样使用implements来明确指定类型和接口之间的关系，因此具体代码只需这样：

```
type PathError struct {
    Op   string
    Path string
    Err  error
}
```

仅供非商业用途或交流学习使用

现在，为了让编译器知道 PathError 可以当一个 error 来传递，还需要关键一步，下面的代码实现了具体的 Error()方法：

```
func (e *PathError) Error() string {
      return e.Op + " " + e.Path + ": " + e.Err.Error()
}
```

关于接口的更多细节，后面相关章节会详细介绍。现在可以直接返回 PathError 变量了，比如在下面的示例代码中，当 syscall.Stat()失败返回 err 时，将该 err 包装到一个 PathError 对象中返回：

```
func Stat(name string) (fi FileInfo, err error) {
      var stat syscall.Stat_t
      err = syscall.Stat(name, &stat)
      if err != nil {
            return nil, &PathError{"stat", name, err}
      }
      return fileInfoFromStat(&stat, name), nil
}
```

不过，很多时候在处理错误时，我们不仅仅满足于打印一句错误信息，而是希望获得更多详细情况，那就需要用到类型转换知识了：

```
      fi, err := os.Stat("a.txt")
      if err != nil {
            if e, ok := err.(*os.PathError); ok && e.Err != nil {
                  // 获取 PathError 类型变量 e 中的其他信息并处理
            }
      }
```

以上就是 Go 语言中 error 类型的简单使用方法。与其他语言中的异常相比，Go 语言的处理相对比较直观、简单。有了错误，就有恢复机制，下面来看 Go 语言的亮点之一：panic 机制。

2. panic

panic（恐慌）是一个内建函数，可以中断原有的控制流程，进入一个令人“恐慌”的流程中（表示程序运行遇到问题，不知道该怎么办，所以很慌张）。当函数 Func1 调用 panic 时，函数的执行就会被中断，但是 Func1 中的 defer 函数会正常执行，然后 Func1 返回到调用它的地方。在调用 Func1 的地方，panic 会继续蔓延，继续向外围扩散，直到 panic 的 goroutine 中所有调用的函数返回，此时程序才会退出。错误信息将被报告，包括在调用 panic()函数时传入的参数，这个过程称为错误处理流程。

仅供非商业用途或交流学习使用

恐慌可以直接调用 panic 产生，也可以由运行时错误产生，例如访问越界的数组。从上面的声明格式来看（参数是一个空接口），panic()函数接收任意类型的数据，比如整型、字符串、对象等。调用方法很简单：

```
panic(404)
panic("network broken")
panic(Error("file not exists"))
```

下面这个函数演示了如何在过程中使用 panic()抛出错误，函数会获取系统的用户名，当用户名不是 root 的时候抛出错误：

```
var user = os.Getenv("USER")

func isRoot() {
      if user != "root" {
            panic("no value for $USER")
      }
}
```

前面产生的错误不影响 defer 的执行，例如：

```
// 示例代码 5-21
package main

func main() {
      defer func() {
            panic("defer panic")

      }()
      panic("test panic")
}
/*
返回：
panic: test panic
      panic: defer panic
......
*/
```

区别使用 panic 和 error 两种方式，导致关键流程出现不可修复性错误的情况使用 panic，其他情况使用 error。

3. recover

recover 是一个内建的函数，可以让进入令人恐慌的流程中的 goroutine 恢复过来。如果当前的 goroutine 陷入恐慌，调用 recover 可以捕获到 panic 的输入值，并且恢复正常的执行。

仅供非商业用途或交流学习使用

由于 recover()函数用于终止错误处理流程，所以一般情况下，recover()仅在 defer 语句的“函数”中有效，以有效截取错误处理流程，recover()只有在 defer 的“函数”内直接调用才会终止错误，否则总是返回 nil。如果在没有发生异常的 goroutine 中明确调用 recover()函数，会导致该 goroutine 所属的进程打印异常信息后直接退出。

```
// 示例代码 5-22
package main

import "fmt"

func main() {
      test()
}
func test() {
      defer func() { // 有效，在 defer 语句的匿名函数中调用
            fmt.Println(recover())
      }()
      defer func() { // 无效，间接调用 recover()，返回 nil
            func() {
                  recover()
            }()
      }()
      defer fmt.Println(recover())/* 无效，recover()相当于直接调用然后被外部函数打印，
                                    返回 nil */
      defer recover()              // 无效，相当于直接调用 recover()，返回 nil
      panic("发生错误")
}
```

以下为一个常见的场景。假设对于 foo()函数的执行心里没底，感觉可能会触发错误处理，或者自己在其中明确加入了按特定条件触发错误处理的语句，那么可以用如下方式在调用代码中截取 recover()：

```
// 示例代码 5-23
package main

import "log"

func main() {
      test()
}
func test() {
      defer func() {
            if r := recover(); r != nil {
                  log.Printf("捕获到的异常：%v", r)
            }
      }()
```

仅供非商业用途或交流学习使用


```
        defer func() {
                panic("第二个错误")
        }()
        panic("第一个错误")
}
/*
返回:
2017/05/03 15:22:44 捕获到的异常: 第二个错误
*/
```

无论 foo()中是否触发了错误处理流程，该匿名 defer 函数都将在函数退出时执行。假如 foo()中触发了错误处理流程，执行 recover()函数将使得该错误处理过程终止。如果错误处理流程被触发，程序传给 panic 函数的参数不为 nil，则该函数还会打印详细的错误信息。

下面这个例子可以检查函数在执行时是否会产生 panic，其中 genErr()函数显然是会产生错误的，然后通过 throwsPanic()函数抛出异常并使用 recover()恢复：

```
// 示例代码5-24
package main

import (
        "fmt"
        "time"
)

func main() {
        // genErr()
        throwsPanic(genErr)
}
func genErr() {
        fmt.Println(time.Now(), "正常的语句")
        defer func() {
                fmt.Println(time.Now(), "defer 正常的语句")
                panic("第二个错误")
        }()
        panic("第一个错误")
}
func throwsPanic(f func()) (b bool) {
        defer func() {
                if r := recover(); r != nil {
                        fmt.Println(time.Now(), "捕获到的异常: ", r)
                        b = true
                }
        }()
        f()
        return
}
```

仅供非商业用途或交流学习使用


```
/*
返回:
.... 15:46:16.973208072 +0800 HKT m=+0.000189341 正常的语句
.... 15:46:16.973306184 +0800 HKT m=+0.000287432 defer 正常的语句
.... 15:46:16.973314041 +0800 HKT m=+0.000295271 捕获到的异常: 第二个错误
*/
```

注意到，虽然“第一个错误”早就发生了，但是在 recover()函数中看不到“第一个错误”的捕获时间，这是因为 recover()只会捕获最后一个错误，而且捕获的时机是在函数最后面，不影响“第一个错误”之后的 defer 语句的执行。

5.4 本章小结

通过本章的学习，相信读者已经能够写出比较简单的完整 Go 语言程序了，目前 Go 语言没有泛型（generic）的概念，也就是说它不支持那种支持多种类型的函数。不过在大部分情况下可以通过接口（interface），特别是空接口与类型选择（type switch）或者通过使用反射（reflection）来实现相似的功能。使用这些技术将导致代码更为复杂、性能更为低下，所以在非常注意性能的场合，最好为每一个类型单独创建一个函数，这样代码可读性更强。

仅供非商业用途或交流学习使用

第6章 复合数据类型

一个稍微复杂的程序往往少不了存储大量的数据，从本章开始将会认识复合数据类型，复合数据类型是可以包含大量条目的数据结构，例如数组、切片和映射等。Go 语言的复合数据类型吸收了很多 Python 语言的优点，相对而言没有 C 语言那么灵活、复杂。

因为引入了切片这种概念使得 Go 语言的代码里并不容易见到数组，切片是构建在数组之上并且提供更强大的能力和便捷的一种数据类型，学好切片有助于在编程过程中灵活处理数据。Go 语言的 3 种复合数据类型可以让开发者管理集合数据，这 3 种数据类型也是 Go 语言核心的一部分，在标准库里被广泛使用。掌握这些数据结构后，用 Go 语言编写程序会变得快速、有趣且十分灵活。

6.1 数组

了解数据结构，一般会从数组开始，因为数组是切片和映射的基础数据结构。理解了数组的工作原理，有助于理解切片和映射提供的优雅、强大的功能。

数组是具有相同类型的一组已经编号且长度固定的数据项序列，这个序列可以是有序的也可以是无序的，组成数组的各个变量称为数组的元素。这种类型可以是任意的原始类型，例如 int、string 或者自定义类型。数组长度必须是一个非负整数的常量（或常量表达式），数组长度也是数组类型的一部分，所以[20]int 和[100]int 不是同一种类型。

6.1.1 声明数组

数组的声明需要指定元素的数据类型，以及存储元素的数量（数组的长度），如下面

仅供非商业用途或交流学习使用

代码所示：

```
// 声明一个包含 5 个元素的整型数组
var array [5]int
```

数组一旦声明，它存储的数据类型和数组长度便都不能修改了。所以如果需要存储更多的元素，必须先创建一个更长的数组，然后再把原来数组里的值复制到新数组里。

Go 语言在声明变量时，都是使用相应类型的零值来初始化变量的，数组也一样。数组初始化时，其每个元素都被初始化为对应类型的零值。例如上面的整型数组里，每个元素都被初始化为 0（整型的零值）。

使用数组字面量可以快速创建并初始化数组，数组字面量可以声明数组里元素的数量，并指定每个元素的值，如下面代码所示：

```
// 声明一个包含 5 个元素的整型数组
// 用具体值初始化每个元素
array := [5]int{10, 20, 30, 40, 50}
```

如果使用...替代数组的长度，Go 语言会根据初始化时数组元素的数量来确定该数组的长度，如下面代码所示：

```
// 声明一个整型数组
// 用具体值初始化每个元素
// 容量由初始化值的数量决定
array := [...]int{10, 20, 30, 40, 50}
```

如果知道数组的长度，还可以给特定下标指定具体值，如下面代码所示：

```
// 声明一个有 5 个元素的数组
// 用具体值初始化索引为 1 和 2 的元素
// 其余元素保持零值
array := [5]int{1: 10, 2: 20}
```

6.1.2 访问与修改

和其他类 C 语言一样，Go 语言数组通过数组下标（索引位置）来读取或者修改数组元素，下标（索引）从 0 开始，第一个元素索引为 0，第二个索引为 1，以此类推。元素的数目（长度或者数组大小）必须是固定的并且在声明数组时就指定的（编译时需要知道数组长度以便分配内存），数组长度最大为 2GB。

1. 访问元素

对于数组 arr[]，第一个元素是 arr[0]，第三个元素是 arr[2]，总体来说下标 i 代表的元

仅供非商业用途或交流学习使用



素是 arr[i]，最后一个元素是 arr1[len(arr1)-1]。下面代码定义一个数组，然后使用 for 循环打印每一个元素的值。

```
// 示例代码 6-1
package main

import "fmt"

func main() {
      arr := [5]int{1, 2, 3, 4}

      for i := 0; i < len(arr); i++ {
            fmt.Printf("Array at index %d is %d\n", i, arr1[i])
      }
}
/*
返回:
Array at index 0 is 1
Array at index 1 is 2
Array at index 2 is 3
Array at index 3 is 4
Array at index 4 is 0
*/
```

2. 修改元素

数组是效率很高的数据结构，因为数组在内存分配中是连续的，要访问数组里某个单独元素，使用[]运算符即可，如下面代码所示：

```
// 声明一个包含 5 个元素的整型数组
// 用具体值初始化每个元素
array := [5]int{10, 20, 30, 40, 50}
// 修改索引为 2 的元素的值
array[2] = 35
```

数组的值也可以是指针，下面声明一个所有元素都是指针的数组，然后使用*运算符就可以访问元素指针所指向的值，如下面代码所示：

```
// 声明包含 5 个元素的指向整数的数组
// 用整型指针初始化索引为 0 和 1 的数组元素
array := [5]*int{0: new(int), 1: new(int)}
// 为索引为 0 和 1 的元素赋值
*array[0] = 10
*array[1] = 20
```

在 Go 语言里，数组是一个类型值，这意味着数组也可以像函数一样用在赋值操作中，变量名代表整个数组，因此同样类型的数组可以赋值给另一个数组，如下面代码所示：

仅供非商业用途或交流学习使用




```
// 声明第一个包含 5 个元素的字符串数组
var array1 [5]string
// 声明第二个包含 5 个元素的字符串数组
// 用颜色初始化数组
array2 := [5]string{"Red", "Blue", "Green", "Yellow", "Pink"}
// 把 array2 的值复制到 array1
array1 = array2
```

复制之后，两个数组的值完全一样。

数组作为一个变量类型，它包括数组长度和每个元素的类型两个部分。只有这两部分都相同的数组才是类型相同的数组，才能互相赋值。看看下面这段代码：

```
// 声明第一个包含 4 个元素的字符串数组
var array1 [4]string
// 声明第二个包含 5 个元素的字符串数组
// 使用颜色初始化数组
array2 := [5]string{"Red", "Blue", "Green", "Yellow", "Pink"}
// 将 array2 复制给 array1
array1 = array2
// Compiler Error:
// cannot use array2 (type [5]string) as type [4]string in assignment
```

与之前的参数传递一样，如果复制数组指针，只会复制指针的值，而不会复制指针所指向的值，如下面代码所示：

```
// 声明第一个包含 3 个元素的指向字符串的指针数组
var array1 [3]*string
// 声明第二个包含 3 个元素的指向字符串的指针数组
// 使用字符串指针初始化这个数组
array2 := [3]*string{new(string), new(string), new(string)}
// 使用颜色为每个元素赋值
*array2[0] = "Red"
*array2[1] = "Blue"
*array2[2] = "Green"
// 将 array2 复制给 array1
array1 = array2
```

在上面的代码中，复制操作之后，两个数组指向同一组字符串。

6.1.3 多维数组

数组本身只有一个维度，但是可以组合多个数组创建多维数组。多维数组多用于管理具有依赖关系的数据（例如坐标系等），下面是声明二维数组的示例代码：

```
// 声明一个二维整型数组，两个维度分别存储 4 个元素和 2 个元素
var array [4][2]int
```

仅供非商业用途或交流学习使用




```
// 使用数组字面量来声明并初始化一个二维整型数组
array := [4][2]int{{10, 11}, {20, 21}, {30, 31}, {40, 41}}
// 声明并初始化外层数组中索引为 1 个和 3 个的元素
array := [4][2]int{1: {20, 21}, 3: {40, 41}}
// 声明并初始化外层数组和内层数组的单个元素
array := [4][2]int{1: {0: 20}, 3: {1: 41}}
```

此时为了访问单个元素，需要组合使用[]运算符，如下面代码所示：

```
// 声明一个 2×2 的二维整型数组
var array [2][2]int
// 设置每个元素的整型值
array[0][0] = 10
array[0][1] = 20
array[1][0] = 30
array[1][1] = 40
```

与一维数组一样，只要类型一致就可以将多维数组互相赋值，如下面代码所示。多维数组类型包括两个部分：每一维度的长度和元素数据类型。

```
// 声明两个不同的二维整型数组
var array1 [2][2]int
var array2 [2][2]int
// 为每个元素赋值
array2[0][0] = 10
array2[0][1] = 20
array2[1][0] = 30
array2[1][1] = 40
// 将 array2 的值复制给 array1
array1 = array2
```

实际上每个数组都是一个单独的值，所以还可以独立复制某个维度，如下面代码所示：

```
// 将 array1 的索引为 1 的维度复制到一个同类型的新数组里
var array3 [2]int = array1[1]
// 将外层数组的索引为 1、内层数组的索引为 0 的整型值复制到新的整型变量里
var value int = array1[1][0]
```

6.1.4　将数组传递给函数

前面讲过，在 Go 语言中数组是一个值类型（value type），所有的值类型变量在赋值和作为参数传递时都将产生一次复制动作。如果直接将数组作为函数的参数，则在函数调用时数组会复制一份作为函数参数。因此，在函数体中无法修改传入的数组的内容，因为函数内操作的只是所传入数组的一个副本。

如此一来，从内存和性能角度看，在函数间传递数组是一个开销很大的操作。特别是

仅供非商业用途或交流学习使用



当这个变量是一个数组时，意味着整个数组，不管有多长，都会完整复制，并传递给函数（别忘了在函数之间传递变量时，默认以值的方式传递）。

为了体验这个操作到底有多消耗性能，我们来创建一个包含 100 万个 int 类型元素的数组。在 64 位架构上，这将需要 800 万字节，即 8MB 的内存。如果声明了这种大小的数组，并将其传递给函数，可以想象是一个非常“重”的操作。

```
// 声明一个需要 8MB 的数组
var array [1e6]int
// 将数组传递给函数 foo
foo(array)
// 函数 foo 接收一个包含 100 万个整型值的数组
func foo(array [1e6]int) {
      // 做些什么
}
```

每次函数 foo()被调用时，必须在栈上分配 8MB 的内存。之后，整个数组的值（大小 8MB）被复制到刚分配的内存里。

虽然 Go 语言会处理这个复制操作，不过其实有一种更好且更高效的方法来处理这个操作，那就是只传入指向数组的指针，这样只需复制 8 字节的数据而不是 8MB 的内存数据到栈上，如下面代码所示：

```
// 分配一个需要 8MB 的数组
var array [1e6]int
// 将数组的地址传递给函数 foo
foo(&array)
// 函数 foo 接收一个指向包含 100 万个整型值的数组的指针
func foo(array *[1e6]int) {
      // 做些什么
}
```

这次函数 foo()接收一个指向 100 万个整型值的数组的指针，现在将数组的地址传入函数，只需在栈上分配 8 字节的内存给指针即可。

这个操作会更有效地利用内存，性能也更好。不过要意识到，因为现在传递的是指针，所以如果改变指针指向的值，会改变共享内存中的值。为了解决这个问题，就需要使用切片来更好地处理这类共享问题了。

6.2 切片

切片是 Go 语言中比较特别的数据结构，这种数据结构更便于使用和管理数据集合。

仅供非商业用途或交流学习使用



切片是围绕动态数组的概念构建的，可以按需自动增长和缩小。切片的动态增长是通过内置函数 append()来实现的，这个函数可以快速且高效地增长切片，也可以通过对切片再次切割，缩小一个切片的大小。因为切片的底层内存也是在连续内存块中分配的，所以切片还能获得索引、迭代以及为垃圾回收优化的好处。

6.2.1　创建数组切片

Go 语言中创建和初始化切片的方法有几种，而能否确定切片的容量是创建切片的关键，它决定了以何种方式创建切片。

1. make 和切片字面量

一种创建切片的方法是使用内置的 make()函数。当使用 make()时，需要传入一个参数，指定切片的长度，如下面代码所示：

```
// 创建一个字符串切片
// 其长度和容量都是 5 个元素
slice := make([]string, 5)
```

如果只指定长度，那么切片的容量和长度相等。也可以分别指定长度和容量，如下面代码所示：

```
// 创建一个整型切片
// 其长度为 3 个元素，容量为 5 个元素
slice := make([]int, 3, 5)
```

分别指定长度和容量时，创建出来的切片的底层数组长度就是创建时指定的容量，但是初始化后并不能访问所有的数组元素。上面代码中的切片可以访问 3 个元素，而底层数组拥有 5 个元素，因此剩余的 2 个元素可以在后期操作中合并到切片，然后才可以通过切片访问这些元素。

基于上面这个切片创建新的切片，新切片会和原有切片共享底层数组，也能通过后期操作来访问多余容量的元素。不过不允许创建容量小于底层数组长度的切片，如下面代码所示：

```
// 创建一个整型切片
// 使其长度大于容量
slice := make([]int, 5, 3)
Compiler Error:
len larger than cap in make([]int)
```

另一种常用的创建切片的方法是使用切片字面量，如下面代码所示：

仅供非商业用途或交流学习使用




```
// 创建字符串切片
// 其长度和容量都是 5 个元素
slice := []string{"Red", "Blue", "Green", "Yellow", "Pink"}
// 创建一个整型切片
// 其长度和容量都是 3 个元素
slice := []int{10, 20, 30}
```

这种方法和创建数组类似，只是不需要指定[]运算符里的值，初始的长度和容量会基于初始化时提供的元素的个数确定。

当使用切片字面量时，可以设置初始长度和容量，要做的就是在初始化时给出所需的长度和容量作为索引（下标）。下面代码中的语法展示了如何创建长度和容量都是 100 个元素的切片：

```
// 创建字符串切片
// 使用空字符串初始化第 100 个元素
slice := []string{99: ""}
```

记住一点，如果在[]运算符里指定了一个值，那么创建的就是数组而不是切片。只有不指定值的时候才会创建切片，如下面代码所示：

```
// 创建有 3 个元素的整型数组
array := [3]int{10, 20, 30}
// 创建长度和容量都是 3 的整型切片
slice := []int{10, 20, 30}
```

2. nil 和空切片

有时程序可能需要声明一个值为 nil 的切片（也称 nil 切片或空切片）。只要在声明时不做任何初始化，就会创建一个 nil 切片，如下面代码所示：

```
// 创建 nil 整型切片
var slice []int
```

在 Go 语言里，nil 切片是很常见的创建切片的方法。nil 切片多用于标准库和内置函数，在需要描述一个目前暂时不存在的切片时，nil 切片十分好用。例如，函数要求返回一个切片但是发生异常的时候，利用初始化，通过声明一个切片可以创建一个 nil 切片：

```
// 使用 make 创建空的整型切片
slice := make([]int, 0)
// 使用切片字面量创建空的整型切片
slice := []int{}
```

nil 切片在底层数组中包含 0 个元素，也没有分配任何存储空间。

此外，nil 切片还可以用来表示空集合。例如，数据库查询返回 0 个查询结果时。nil 切片和普通切片一样，调用内置函数 append、len 和 cap 的效果都是一样的。

仅供非商业用途或交流学习使用



6.2.2　切片的使用

现在知道了什么是切片，也知道如何创建切片，下面来看看如何在程序里使用切片。

1. 赋值和分割

给切片的某个元素赋值和给数组的某个元素赋值在方法上完全一样，使用[]运算符就可以改变某个元素的值。

```
// 创建一个整型切片
// 其容量和长度都是 5 个元素
slice := []int{10, 20, 30, 40, 50}
// 改变索引为 1 的元素的值
slice[1] = 25
```

切片之所以被称为切片，是因为创建一个新的切片就是把底层数组切出一部分。例如：

```
// 创建一个整型切片
// 其长度和容量都是 5 个元素
slice := []int{10, 20, 30, 40, 50}
// 创建一个新切片
// 其长度为 2 个元素，容量为 4 个元素
newSlice := slice[1:3]
// 使用 3 个索引创建切片
newSlice2 := slice[2:3:4]
```

执行上面的代码后，就有了两个切片，它们共享同一段底层数组，但通过不同的切片会看到底层数组的不同部分。

第一个切片 slice 能够访问底层数组全部 5 个元素的容量，不过之后的 newSlice 就访问不到了。对于 newSlice，底层数组的容量只有 4 个元素。newSlice 无法访问到它所指向的底层数组的第一个元素之前的部分。换而言之，对 newSlice 来说，之前的那些元素是不存在的。

一个常见的描述是，对底层数组容量是 k 的切片 slice[i:j:k]来说，长度为 j-i，容量为 k-i。

需要记住的是，现在两个切片共享同一个底层数组，如果一个切片修改了该底层数组的共享部分，另一个切片也会被影响：

```
// 创建一个整型切片
// 其长度和容量都是 5 个元素
slice := []int{10, 20, 30, 40, 50}
// 创建一个新切片
// 其长度是 2 个元素，容量是 4 个元素
```

仅供非商业用途或交流学习使用




```
newSlice := slice[1:3]
// 修改 newSlice 索引为 1 的元素
// 同时也修改了原来的 slice 的索引为 2 的元素
newSlice[1] = 35
```

把 35 赋值给 newSlice 的第二个元素(索引为 1 的元素)的同时也是在修改原来的 slice 的第 3 个元素（索引为 2 的元素)。切片只能访问到其长度内的元素，试图访问超出其长度的元素将会导致语言运行时异常：

```
// 创建一个整型切片
// 其长度和容量都是 5 个元素
slice := []int{10, 20, 30, 40, 50}
// 创建一个新切片
// 其长度为 2 个元素，容量为 4 个元素
newSlice := slice[1:3]
// 修改 newSlice 索引为 3 的元素
// 这个元素对于 newSlice 来说并不存在
newSlice[3] = 45
// Runtime Exception:
// panic: runtime error: index out of range
```

在使用与切片的容量相关联的元素前，必须将其合并到切片的长度里，只能用于增长切片。总的来说，切片有额外的容量是很好的，但是如果不能把这些容量合并到切片的长度里，这些容量就没有用处。当然 Go 语言是由内置函数来解决这个问题的，append()就是专门来做这种合并的内置函数。

2. 切片扩容

相对于数组而言，使用切片的一个好处是可以按需增加数据集合的容量。Go 语言内置的 append()函数会处理增加长度时的所有操作细节。

要使用 append()，需要一个被操作的切片和一个要追加的值：

```
// 创建一个整型切片
// 其长度和容量都是 5 个元素
slice := []int{10, 20, 30, 40, 50}
// 创建一个新切片
// 其长度为 2 个元素，容量为 4 个元素
newSlice := slice[1:3]
// 使用原有的容量来分配一个新元素
// 将新元素赋值为 60
newSlice = append(newSlice, 60)
```

当 append()被调用时会返回一个包含修改结果的新切片。函数 append()只能增加新切片的长度，而容量有可能会改变，也可能不会改变，这取决于被操作的切片的可用容量。

仅供非商业用途或交流学习使用

因为 newSlice 在底层数组里还有额外的容量可用，append()操作将可用的元素合并到切片的长度，并对其进行赋值。由于和原始的 slice 共享同一个底层数组，slice 中索引为 3 的元素的值也被改动了。

如果切片的底层数组没有足够的可用容量，append()函数会创建一个新的底层数组，将被引用的现有的值复制到新数组里，再追加新的值：

```
// 创建一个整型切片
// 其长度和容量都是 4 个元素
slice := []int{10, 20, 30, 40}
// 向切片追加一个新元素
// 将新元素赋值为 50
newSlice := append(slice, 50)
```

在这个 append()操作完成后，newSlice 拥有一个全新的底层数组，这个数组的容量是原来的两倍。

函数 append()会智能地处理底层数组的容量，使其增长。在切片的容量小于 1000 个元素时，总是会成倍地增加容量。一旦元素个数超过 1000，容量的增长因子会设为 1.25，也就是会每次增加 25%的容量（随着语言的演化，这种增长算法可能会有所改变）。

3. 遍历切片

既然切片是一个集合，那么自然可以迭代其中的元素。前面我们已经提到过，Go 语言有个特殊的关键字 range，它可以配合关键字 for 来迭代切片里的元素：

```
// 使用 for range 迭代切片
// 创建一个整型切片，长度和容量都是 4 个元素
slice := []int{10, 20, 30, 40}
// 迭代元素并显示
for index, value := range slice {
    fmt.Printf("index : %d value: %d\n", index, value)
}
// index : 0 value: 10
// index : 1 value: 20
// index : 2 value: 30
// index : 3 value: 40
```

和数组类似，迭代切片时，range 会返回两个值。第一个值是当前迭代到的索引位置，第二个值是该位置对应元素值的一份副本。

注意，range 创建的是每个元素的副本，而不是返回对该元素的引用，如下面代码所示。如果使用该值变量的地址作为指向每个元素的指针，就会造成错误。

仅供非商业用途或交流学习使用


```
// 创建一个整型切片
// 其长度和容量都是 4 个元素
slice := []int{10, 20, 30, 40}
// 迭代每个元素，并显示值和地址
for index, value := range slice {
      fmt.Printf("Value: %d Value-Addr: %X ElemAddr: %X\n", value, &value, &slice[i
ndex])
}
/*
返回：
Value: 10 Value-Addr: 10500168 ElemAddr: 1052E100
Value: 20 Value-Addr: 10500168 ElemAddr: 1052E104
Value: 30 Value-Addr: 10500168 ElemAddr: 1052E108
Value: 40 Value-Addr: 10500168 ElemAddr: 1052E10C
*/
```

上面的 Value-Addr 之所以全部一样，是因为 range 执行时返回的这个 value 变量实际上是同一个变量，被依次不断赋值输出，所以 value 的地址总是相同的。因此要想获取每个元素的地址，可以使用切片变量和索引值。如果不用索引值，可以使用空白占位字符来忽略这个值：

```
// 创建一个整型切片
// 其长度和容量都是 4 个元素
slice := []int{10, 20, 30, 40}
// 迭代每个元素，并显示其值
for _, value := range slice {
      fmt.Printf("Value: %d\n", value)
}
/*
返回：
Value: 10
Value: 20
Value: 30
Value: 40
*/
```

关键字 range 总是从切片头部开始迭代的。如果想对迭代做更多的控制，依旧可以使用传统的 for 循环：

```
// 创建一个整型切片
// 其长度和容量都是 4 个元素
slice := []int{10, 20, 30, 40}
// 从第三个元素开始迭代每个元素
for index := 2; index < len(slice); index++ {
      fmt.Printf("Index: %d Value: %d\n", index, slice[index])
}
/*
```

仅供非商业用途或交流学习使用


```
返回:
Index: 2 Value: 30
Index: 3 Value: 40
*/
```

上面代码遇到了一个内置函数，在 Go 语言中，有两个特殊的内置函数 len()和 cap()，可以用于处理数组、切片和通道。对于切片，函数 len()返回切片的长度，函数 cap()返回切片的容量。在上面代码里，使用函数 len()来决定什么时候停止对切片的迭代。

现在知道了如何创建和使用切片。可以组合多个切片成为多维切片，并对其进行迭代。

4．限制容量

在创建切片时，还可以使用之前没有提及的第三个索引选项。第三个索引可以用来控制新切片的容量，其目的并不是要增加容量，而是要限制容量。可以看到，允许限制新切片的容量为底层数组提供了一定的保护，可以更好地控制追加操作。

让我们看看一个包含 5 个元素的字符串切片，这个切片包含了本地超市能找到的水果名字。

```
// 创建字符串切片
// 其长度和容量都是 5 个元素
source := []string{"Apple", "Orange", "Plum", "Banana", "Grape"}
```

现在，让我们试着用第三个索引选项来完成切片操作：

```
// 将第三个元素做切片，并限制容量
// 其长度为 1 个元素，容量为 2 个元素
slice := source[2:3:4]
```

这个切片操作执行后，新切片里从底层数组引用了 1 个元素，容量是 2 个元素。具体来说，新切片引用了 Plum 元素，并将容量扩展到 Banana 元素。可以应用之前的方法来计算新切片的长度和容量：对于 slice[i:j:k]([2:3:4])，长度为 j–i(即 3-2=1)，容量为 k–i(4-2=2)。

和之前一样，第一个值表示新切片开始的元素的索引位置，这个例子中是 2。第二个值表示开始的索引位置（2）加上希望包括的元素的个数（1），2+1 的结果是 3，所以第二个值就是 3。为了设置容量，从索引位置 2 开始，加上希望容量中包含的元素的个数（2），就得到了第三个值 4。

如果试图设置的容量比可用的容量还大，就会得到一个语言运行时错误：

```
// 这个切片操作试图设置容量为 4 个元素
// 这比可用的容量大
slice := source[2:3:6]
```

```
// Runtime Error:
// panic: runtime error: slice bounds out of range
```

我们之前讨论过，内置函数 append()会首先使用可用容量。一旦没有可用容量，会分配一个新的底层数组，这导致很容易忘记切片间正在共享同一个底层数组，一旦发生这种情况，对切片进行修改，很可能会导致随机且奇怪的问题。

对切片内容的修改会影响多个切片，却很难找到问题的原因。如果在创建切片时设置切片的容量和长度一样，就可以强制让新切片的第一个 append()操作创建新的底层数组，与原有的底层数组分离。

新切片与原有的底层数组分离后，可以安全地进行后续修改：

```
// 创建字符串切片
// 其长度和容量都是 5 个元素
source := []string{"Apple", "Orange", "Plum", "Banana", "Grape"}
// 对第三个元素做切片，并限制容量
// 其长度和容量都是 1 个元素
slice := source[2:3:3]
// 向 slice 追加新字符串
slice = append(slice, "Kiwi")
```

如果不加第三个索引，由于剩余的所有容量都属于 slice，向 slice 追加 Kiwi 元素会改变原有底层数组索引为 3 的元素的值 Banana，不过在上面代码中限制了 slice 的容量为 1。当第一次对 slice 调用 append()时，会创建一个新的底层数组，这个数组包括 2 个元素，并将水果 Plum 复制进来，再追加新水果 Kiwi，并返回一个引用了这个底层数组的新切片。

因为新的切片 slice 拥有了自己的底层数组，所以杜绝了可能发生的问题。我们可以继续向新切片追加水果，而不用担心会不小心修改了其他切片里的水果。同时，也保持了为切片申请新的底层数组的便捷性。

内置函数 append()也是一个可变参数的函数，这意味着可以在一次调用中传递多个追加的值。如果使用...运算符，可以将一个切片的所有元素追加到另一个切片里：

```
// 创建两个切片，并分别用两个整数进行初始化
s1 := []int{1, 2}
s2 := []int{3, 4}
// 将两个切片追加在一起，并显示结果
fmt.Printf("%v\n", append(s1, s2...))
/*
返回：
[1 2 3 4]
*/
```

就像通过输出看到的那样，切片 s2 里的所有值都追加到了切片 s1 的后面。使用 Printf 显示 append()函数返回的新切片的值。

6.2.3　多维切片

和数组一样，切片是一维的。不过，和之前对数组的讨论一样，可以组合多个切片形成多维切片：

```
// 创建一个整型切片的切片
slice := [][]int{{10}, {100, 200}}
```

一个包含两个元素的外层切片，每个元素包含一个内层的整型切片。外层的切片包括两个元素，每个元素都是一个切片。

第一个元素中的切片使用单个整数 10 来初始化，第二个元素中的切片包括两个整数，即 100 和 200，这种组合可以让用户创建非常复杂且强大的数据结构。已经学过的关于内置函数 append()的规则也可以应用到组合后的切片上：

```
// 创建一个整型切片的切片
slice := [][]int{{10}, {100, 200}}
// 为第一个切片追加值为 20 的元素
slice[0] = append(slice[0], 20)
```

Go 语言里使用 append()函数处理追加的方式很简明：先增长切片，再将新的整型切片赋值给外层切片的第一个元素。执行以上代码后，会为新的整型切片分配新的底层数组，然后将切片复制到外层切片的索引为 0 的元素中。

即便是这么简单的多维切片，操作时也会涉及众多布局和值。看起来在函数间像这样传递数据结构也会很复杂，不过切片本身结构很简单，可以以很小的成本在函数间传递。

6.2.4　将切片传递给函数

在函数间传递切片就是要在函数间以值的方式传递切片。由于切片的尺寸很小，在函数间复制和传递切片成本也很低。让我们创建一个大切片，并将这个切片以值的方式传递给函数 foo()：

```
// 分配包含 100 万个整型值的切片
slice := make([]int, 1e6)
// 将 slice 传递到函数 foo
slice = foo(slice)
// 函数 foo 接收一个整型切片，并返回这个切片
```

```
func foo(slice []int) []int {
    ...
    return slice
}
```

在 64 位架构的机器上，一个切片需要 24B 的内存：指针字段需要 8B，长度和容量字段分别需要 8B。由于与切片关联的数据包含在底层数组里，不属于切片本身，所以将切片复制到任意函数的时候，对底层数组大小都不会有影响。复制时只会复制切片本身，不会涉及底层数组。

在函数间传递 24B 的数据会非常快速、简单，这也是切片效率高的地方。不需要传递指针和处理复杂的语法，只需复制切片，按想要的方式修改数据，然后传递回一份新的切片副本。

6.3 映射

映射是一种数据结构，用于存储一系列无序的键值对，映射基于键来存储值。映射功能强大的地方是，能够基于键快速检索数据。键就像索引一样，指向与该键关联的值。

与 C++、Java 不一样，Go 语言使用映射（map）不需要引入任何库，而 C++、Java 需要先引用相应的库，因此 Go 语言的映射使用起来更加方便。

6.3.1 映射的实现

因为映射也是一个数据集合，所以也可以使用类似处理数组和切片的方式来迭代映射中的元素。但映射是无序集合，所以即使以同样的顺序保存键值对，每次迭代映射时，元素顺序也可能不一样。无序的原因是映射的实现使用了散列表。

Go 语言中的 map 在底层是用哈希（hash）表实现的，在$GOROOT/src/pkg/runtime/hashmap.go 可以查看它的实现细节。Go 语言的 map 是一个 hash 数组列表，而不是像 C++ 一样使用红黑树，与传统的 hashmap 一样，Go 语言的 map 由一个个 bucket 组成，如图 6-1 所示。

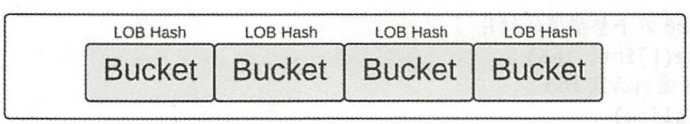


图 6-1　hashmap 内部结构

此列表中的每一个元素都被称为 bucket 的结构体，每个 bucket 可以保存 8 个键值对，所有元素将被 hash 算法填入到数组的 bucket 中，bucket 填满后，将通过一个 overflow 指针来扩展一个 bucket，从而形成链表，以此解决 hash 冲突的问题。简单来说，这个 map 就是一个 bucket 指针型的一维数组，而每个 bucket 指针下面则不定长，可能挂着 bucket 指针 list，也可能只有一个，视 hash 冲突而定。

6.3.2　映射的创建

Go 语言创建并初始化映射有很多种方法，使用内置的 make 函数或者使用映射字面量都是最常见的办法：

```
// 创建一个映射，键的类型是 string，值的类型是 int
dict := make(map[string]int)
// 创建一个映射，键和值的类型都是 string
// 使用两个键值对初始化映射
dict := map[string]string{"Red": "#da1337", "Orange": "#e95a22"}
```

使用映射字面量是更常用的方法，映射的初始长度会根据初始化时指定的键值对的数量来确定。

映射的键可以是任何值，这个值的类型并不限制，内置的类型或者结构类型都可以，不过需要确定这个值可以使用==运算符做比较。需要注意的是，切片、函数以及包含切片的结构类型由于具有引用语义，均不能作为映射的键，使用这些类型会造成编译错误：

```
// 创建一个映射，使用字符串切片作为映射的键
dict := map[[]string]int{}
// Compiler Exception:
// invalid map key type []string
```

可以使用切片作为映射的值：

```
// 创建一个映射，使用字符串切片作为值
dict := map[int][]string{}
```

在使用一个映射键对应一组数据时，会非常有用。

6.3.3　映射的使用

1. 元素赋值

通过指定适当类型的键并给这个键赋一个值就完成了映射的键值对赋值：

```
// 创建一个空映射，用来存储颜色以及颜色对应的十六进制代码
colors := map[string]string{}
// 将 Red 的代码加入到映射
colors["Red"] = "#DA1337"
```

与切片类似，通过声明一个未初始化的映射可以创建一个值为 nil 的映射（称为 nil 映射）。nil 映射不能用于存储键值对，否则，会产生一个语言运行时错误：

```
// 通过声明映射创建一个 nil 映射
var colors map[string]string
// 将 Red 的代码加入到映射
colors["Red"] = "#da1337"
// Runtime Error:
// panic: runtime error: assignment to entry in nil map
```

查找映射里是否存在某个键是映射的一个基本操作，这个操作往往需要用户写一些逻辑代码，根据逻辑代码确定是否完成了某个操作或者映射里是否缓存了某些数据。查找操作也可以用来比较两个映射，确定哪些键值对互相匹配，哪些键值对不匹配。

2. 查找与遍历

从映射取值时有两种方式。第一种方式是，获得值以及一个表示这个键是否存在的标志：

```
// 获取键 Blue 对应的值
value, exists := colors["Blue"]
// 这个键存在吗?
if exists {
      fmt.Println(value)
}
```

另一种方式是，只返回键对应的值，再判断这个值是否为零值，以此来确定键是否存在。显然，这种方式只能用在映射存储的值都是非零值的情况：

```
// 获取键 Blue 对应的值
value := colors["Blue"]
// 这个键存在吗?
if value != "" {
      fmt.Println(value)
}
```

在 Go 语言里，通过键来索引映射时，即便这个键不存在也会返回该值对应的类型的零值。

和迭代数组或切片一样，使用关键字 range 可以迭代映射里的所有值：

```
// 创建一个映射，存储颜色以及颜色对应的十六进制代码
colors := map[string]string {
  "AliceBlue": "#f0f8ff",
  "Coral": "#ff7F50",
  "DarkGray": "#a9a9a9",
  "ForestGreen": "#228b22",
}
// 显示映射里的所有颜色
for key, value := range colors {
      fmt.Printf("Key: %s Value: %s\n", key, value)
}
```

3. 元素删除

Go 语言提供了一个内置函数 delete()用于删除容器内的元素。下面是用 delete()函数删除 map 内的元素的一个例子：

```
delete(myMap, "1234")
```

上面的代码将从 myMap 中删除键为 1234 的键值对。如果 1234 这个键不存在，那么这个调用将什么都不发生，也不会有什么副作用。但是如果传入的 map 变量的值是 nil，该调用将导致程序抛出异常（panic）。

以上面的 range 为例，对映射来说，range 返回的不是索引和值，而是键值对。如果想把一个键值对从映射里删除，可以使用内置的 delete 函数：

```
// 删除键为 Coral 的键值对
delete(colors, "Coral")
// 显示映射里的所有颜色
for key, value := range colors {
      fmt.Printf("Key: %s Value: %s\n", key, value)
}
```

这次在迭代映射时，颜色 Coral 不会显示在屏幕上。

6.3.4　将映射传递给函数

在函数间传递映射并不会制造出该映射的一个副本。实际上，当传递映射给一个函数，并对这个映射做了修改时，所有对这个映射的引用都会察觉到这个修改：

```
// 示例代码 6-2
package main

import (
      "fmt"
)
```

```
func main() {
	// 创建一个映射，存储颜色以及颜色对应的十六进制代码
	colors := map[string]string{
		"AliceBlue":   "#f0f8ff",
		"Coral":       "#ff7F50",
		"DarkGray":    "#a9a9a9",
		"ForestGreen": "#228b22",
	}
	// 显示映射里的所有颜色
	for key, value := range colors {
		fmt.Printf("Key: %s Value: %s\n", key, value)
	}
	// 调用函数来移除指定的键
	removeColor(colors, "Coral")

	// 显示映射里的所有颜色
	for key, value := range colors {
		fmt.Printf("Key: %s Value: %s\n", key, value)
	}
}

// removeColor 将指定映射里的键删除
func removeColor(colors map[string]string, key string) {
	delete(colors, key)
}
/*
返回:
Key: AliceBlue Value: #F0F8FF
Key: Coral Value: #FF7F50
Key: DarkGray Value: #A9A9A9
Key: ForestGreen Value: #228B22
Key: AliceBlue Value: #F0F8FF
Key: DarkGray Value: #A9A9A9
Key: ForestGreen Value: #228B22
*/
```

可以看到，在调用了 removeColor 之后，main 函数引用的映射中不再有 Coral 颜色了。这个特性和切片类似，保证可以用很小的成本来复制映射。

6.4 本章小结

本章介绍了常用的复合数据类型，其中数组是构造切片和映射的基石。Go 语言里切片经常用来处理数据的集合，映射用来处理具有键值对结构的数据。

内置函数 make 可以创建切片和映射，并指定原始的长度和容量。也可以直接使用切片和映射字面量，或者使用字面量作为变量的初始值。切片有容量限制，不过可以使用内置的 append 函数扩展容量。映射的增长没有容量或者任何限制。

内置函数 len 可以用来获取切片或者映射的长度，内置函数 cap 只能用于切片，通过组合，可以创建多维数组和多维切片，也可以使用切片或者其他映射作为映射的值，但是切片不能用作映射的键。将切片或者映射传递给函数成本很小，并且不会复制底层的数据结构。

第7章 包

现如今即便是个人开发的一般程序，其包含的函数可能超过了 1 万个，这些函数代码一般都由他人编写并打包为“包”或者“模块”的形式，并通过相关社区分发，最后由软件作者通过调用这些“包”或者“模块”的函数来更高效地完成开发。因此在大部分时候，作者只会用到其中很小的一部分函数，但是通过“包”或者“模块”的形式重用代码使得编程开发变得更加轻松，这也是当今大部分编程语言都有自己的包管理工具和包分发渠道的原因。

在前面的章节中，几乎每个例子都使用到了 Go 语言包，例如像 fmt、os 等这样具有常用功能的内置包在 Go 语言中有一百多个，我们习惯称为标准包（标准库），这些标准包大部分都内置到 Go 语言本身。（完整列表可以在 Go Walker 中查看，或者使用 go list std 命令查看标准包列表。）

在本章的学习中，将包含以下知识点：

- 包结构认识
- 包的使用基础
- 常用标准包详解
- 使用自定义包

7.1 包的基础

在大部分编程语言中都存在“包”的概念，而任何一种包设计的目的都是为了简化大型软件设计和维护的工作，实际上包是函数和数据的集合，通过把一组有相关特性的函数

和数据放进一个单元中，方便使用和更新相应的模块。这种包系统的设计使得每一个模块（包）都与程序、其他单元（其他包）保持一定的独立性，这使得每一个模块（包）可以被应用到不同的程序部分中，当然也包括其他程序中，甚至可以通过社区分发渠道流通到世界各地的项目中被不断重复利用，不仅降低了项目模块之间的耦合度，也提高了整体的开发效率。

在编写代码的过程中，不同模块（包）之间为了实现某一个类似的功能可能会采用相同的名字去命名一个函数，如果一个软件开发过程中需要同时使用两个模块（包），就会在调用函数时产生歧义。为了解决这个问题，Go 语言引入了命名空间的概念，让每个包都定义一个命名空间，用于内部标识符的访问。因为每个命名空间关联一个特定的包，这使得我们在调用类型、函数时有了独一无二的简短明了的名字，避免在使用它们的时候产生命名冲突。

为了提高包的独立性以及安全性，Go 语言的包可以通过控制包内名字的可见性来实现包的封装，通过限制包成员的可见性、隐藏具体的实现过程可以极大提高软件的安全性，同时开发人员调用时也不必关心其实现过程，直接使用包的 API，另一方面也允许包的维护者在不影响包的用户使用的前提下调整包的内部实现。通过限定包内函数与变量的可见性来约束用户操作，只允许用户通过特定的函数来访问和控制包内部的变量，这样不仅可以提高安全性（变量一致）还可以保持并发操作时的互斥约束。

与大部分编译语言类似，在 Go 语言中，当改动了一个源文件时，就必须重新编译该源文件，以及它对应的包和所有依赖该包的其他包。即使是从头构建，Go 语言编译器的编译速度也明显快于绝大部分编译语言。如此优异的编译速度主要得益于其包设计的三个特性。

（1）显式声明：所有导入的包必须在每个文件的开头显式声明，这样编译器就没有必要读取和分析整个源文件来判断包的依赖关系。

（2）无环依赖：禁止包的环状依赖，因为没有循环依赖，包的依赖关系形成一个有向无环图，每个包可以被独立编译，而且很可能是被并发编译。

（3）无须遍历：编译后包的目标文件不仅记录包本身的导出信息，同时还记录了包的依赖关系。因此，在编译一个包的时候，编译器只需要读取每个直接导入包的目标文件，而不需要遍历所有依赖的文件，毕竟很多都是重复的间接依赖。

7.1.1 包的结构

在前面的学习中就可以知道 Go 语言编译工具对源码目录有很严格的要求，每个工作空间（workspace）必须由 bin、pkg、src 三个目录组成。bin 目录主要存放可执行文件；pkg 目录存放编译好的库文件，主要是*.a 文件；src 目录主要存放 Go 语言的源文件。

```
.
|-- bin
|   |-- goimports
|   `-- gophernotes
|-- pkg
|   `-- linux_amd64
|       |-- github.com
|       |   |-- gopherds
|       |   |   `-- gophernotes
|       |   |       |-- internal
|       |   |       |   `-- repl.a
|       |   ... ...
`-- src
    |-- github.com
    |   |-- gopherds
    ... ...
```

1. 工作空间

因为 Go 语言采用了工作空间这种方式来管理本地代码，这与大部分编程语言不一样，因此这里解释一下 GOROOT 和 GOPATH 之间的关系。首先，显而易见的一点就是 GOROOT 是一个全局并且唯一的变量，用于指定存放 Go 语言本身的目录路径（安装路径）；而 GOPATH 是一个工作空间的变量，它可以有很多个（用;号分隔），用于指定工作空间的目录路径。例如：

```
GOPATH=$HOME/workspace/golib:$HOME/projects/go
```

通常 go get 会使用第一个工作空间保存下载的第三方库（包），在开发时不管是哪一个工作空间下载的包都可以在任意工作空间使用。需要注意的一点就是尽量不要把 GOROOT 和 GOPATH 设置为同一个路径。

包的源代码书写与正常的 Go 语言没有区别，文件必须是 UTF-8 格式，编写规范与本书强调的编程规范一致。

2. 包的源文件

包的代码必须全部都放在包中，并且源文件头部都必须一致使用 package <name>的语

句进行声明。Go 语言包可以由多个文件组成，所以文件名不需要与包名一致，包名建议使用小写字符。包名类似命名空间（namespace），与包所在目录、编译文件名无关，目录名尽量不要使用保留名称（main、all、std），对于可执行文件必须包含 package main 以及入口函数 main。

Go 语言使用名称首字母大小写来判断一个对象（全局变量、全局常量、类型、结构字段、函数、方法）的访问权限，对于包而言同样如此。包中成员名称首字母大小写决定了该成员的访问权限。首字母大写，可被包外访问，即为 public（公开的）；首字母小写，则仅包内成员可以访问，即为 internal（内部的）。

与大部分现代编程语言一样，Go 语言同样支持使用 UTF-8 字符来命名对象，因此关于“大写”这个概念不限于 US ASCII，它被扩展到了所有大小写字母表（包括拉丁文、希腊文、斯拉夫文、亚美尼亚文和埃及古文等）。汉字一般没有大小写的概念（除了汉字数字），因此如果使用汉字作为一个函数的名称，该函数默认是私有的，需要在汉字前面加上一个大写字母才能使其变为公有函数。

3. 包的声明

上面提到每一个包内源文件都需要在开头声明所在包，这其实就是包的声明。包的声明对于包内而言主要用于源文件编译时能够为编译器指明哪些是包的源代码；对于包外而言，在导入包的时候可以使用“包名.函数名”的方式使用包内函数。

对于包名相同的情况，例如 math/rand 包和 crypto/rand 包的包名都是 rand，Go 语言也有相应的办法解决，在下一节导入包时将会介绍。

关于包的声明有一个例外，那就是包编译后是一个可执行程序时，会使用 package main 的方式声明 main 包，这时候 main 包本身的导入路径是无关紧要的，这个名字实际是给 go build 构建命令一个信息，这个包编译完之后必须调用连接器生成一个可执行程序。（本章暂不讨论测试包的情况。）

7.1.2 包的导入

如前所述，使用包成员之前先要导入包。导入包的关键字是 import，因为 Go 语言包不能形成环形依赖，如果遇到导入包循环依赖的情况，Go 语言的构建工具将返回错误。一般而言对于直接从分发渠道下载回来的包都不会轻易产生依赖环：

```
import "相对目录/包主文件名"
```

相对目录是指从/pkg/开始的子目录，以标准库为例：

```
import "fmt" // 对应/usr/local/go/pkg/linux_amd64/fmt.a
import "os/exec" // 对应/usr/local/go/pkg/linux_amd64/os/exec.a
```

除了一行一个包的方式导入，还可以使用一条语句导入多个包的写法：

```
import (
      "fmt"
      "os/exec"
)
```

1. 导入声明

在上一节中有一个问题没有解决，就是同名包导入时会有冲突。虽然包的命名空间解决了函数重名的情况，但是没有避免包重名的情况，因此 Go 语言在导入包时可以对包名做重定向，以解决包名冲突的情况。例如下面的几种例子。

```
import   "crypto/rand" // 默认模式: rand.Function
import R "crypto/rand" // 包重命名: R.Function
import . "crypto/rand" // 简便模式: Function
import _ "crypto/rand" // 匿名导入: 仅让该包执行初始化函数
```

另一种写法：

```
import (
   "crypto/rand"
   mrand "math/rand" // 包重命名
)
```

注意：

（1）Go 语言不允许导入包却又不使用，如果导入的包并未使用，在编译时会被视为错误（不包括 "import _"）。

（2）包的重命名不仅可以用于解决包名冲突，还可以解决包名过长、避免与变量或常量名称冲突等情况。

除了以上比较常见的包导入方式，还有子包导入方式以及自定义路径导入包方式。其中对于当前目录下的子包，除使用默认完整导入路径外，还可使用相对路径的方式。

```
.
└── src
    └── test
        ├── main.go
        └── test2
            └── test.go
```

如上面的目录结构，可以在 main.go 文件中使用下面的方式导入 test2 这个包：

```
import "test/test2" // 一般使用这种方式导入
import "./test2" // 也可以使用相对目录，但这种方式导入的包仅对 Go 语言有效
```

如果在一个文件中导入的包比较多，为了管理源代码中导入的包，还可以为导入的包分组。分组是通过空行来分隔的，例如：

```
import (
    "fmt"
    "html/template"
    "os"

    "golang.org/x/net/html"
    "golang.org/x/net/ipv4"
)
```

我们知道 Go 语言编译的时候会格式化代码，因此导入包的顺序并不需要我们调整，编译时会自动按字母排序。可以调整的只有包分组，同样每一个分组内的包在编译时会被格式化为按字母排序。

2. 导入路径

当前 Go 语言的规范并没有强制包的导入路径字符串的格式，导入路径由构建工具来解释。但如果打算分享或发布自己编写的包，那么最好使用全球唯一的导入路径。

这主要是为了避免导入路径冲突，因此有一个约定俗成的路径格式是：所有非标准库包的导入路径以所在组织的互联网域名为前缀，这样一来就有了一个独一无二的路径，另一方面也有利于包的检索。

例如下面导入的包中就有两个使用了互联网域名为前缀：

```
import (
    "fmt"
    "math/rand"
    "encoding/json"
    "golang.org/x/net/html"
    "github.com/go-sql-driver/mysql"
)
```

3. 自定义路径

在上一节中使用了一种域名为前缀的导入路径，对于编译器来说，只有较为流行的代码托管站点才可以直接使用这种路径。对于一些个人站点（例如企业自己搭建的私有 GitLab 仓库），为了可以更方便使用这种方式导入就需要告诉编译器这是一个包代码链接。

我们有三种方式实现这个功能，一是直接在包链接中加上 VCS 格式，目前支持的格式有：

```
Bazaar      .bzr
Git         .git
Mercurial   .hg
Subversion  .svn
```

例如：

```
import "example.org/user/foo.git"
```

第二种办法是针对没有提供版本控制符的链接，go get 甚至不知道应该如何下载代码的情况，例如下面这种链接：

```
example.org/repo/foo
```

这个时候就需要在网页中加入一句标签：

```
<meta name="go-import" content="import-prefix vcs repo-root">
```

然后就可以使用链接导入：

```
import "example.org/pkg/foo"
```

第三种情况是重定向网页链接，例如下面的情况，go get 访问链接时会被重定向到 example.org/r/p/exproj：

```
<meta name="go-import" content="example.org git https://example.org/r/p/exproj">
```

如果没有服务器还可以使用 Go 语言搭建一个简单的本地服务器：

```
package main
import (
    "fmt"
    "net/http"
)
func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprint(w, `<meta name="go-import"
    content="example.com/zuolan/test git https://github.com/zuolan/test">`)
}
func main() {
    http.HandleFunc("/zuolan/test", handler)
    http.ListenAndServe(":80", nil)
}
```

保存为 server.go，然后编译执行，就可以实现把 example.com/zuolan/test 重定向到 github.com/zuolan/test。

4. 匿名导入

如果只是导入一个包而并不使用导入的包将会导致一个编译错误。但是有时候我们只是想利用导入包而产生的副作用：它会计算包级变量的初始化表达式和执行导入包的 init 初始化函数。这时候需要抑制“unused import”编译错误，可以用下画线来重命名导入的包。像往常一样，下画线为空白标识符，并不能被访问：

```
import _ "image/png" // PNG 解码器
```

这个被称为包的匿名导入，它通常用来实现一个编译时机制，然后通过在 main 主程序入口选择性地导入附加的包。首先，让我们看看如何使用该特性，然后再看看它是如何工作的。

标准库的 image 图像包包含了一个 Decode 函数，用于从 io.Reader 接口读取数据并解码图像，它调用底层注册的图像解码器来完成任务，然后返回 image.Image 类型的图像。使用 image.Decode 很容易编写一个图像格式的转换工具，读取一种格式的图像，然后编码为另一种图像格式：

```
// 示例代码 7-1
// jpeg 包从标准输入读取 PNG 图像，并将其作为 JPEG 图像写入标准输出
package main

import (
    "fmt"
    "image"
    "image/jpeg"
    _ "image/png" // PNG 解码器
    "io"
    "os"
)

func main() {
    if err := toJPEG(os.Stdin, os.Stdout); err != nil {
        fmt.Fprintf(os.Stderr, "jpeg: %v\n", err)
        os.Exit(1)
    }
}

func toJPEG(in io.Reader, out io.Writer) error {
    img, kind, err := image.Decode(in)
    if err != nil {
        return err
    }
    fmt.Fprintln(os.Stderr, "Input format =", kind)
    return jpeg.Encode(out, img, &jpeg.Options{Quality: 95})
}
```

如果将输出导入一个程序的标准输入，它将解码输入的 PNG 格式图像，然后转换为 JPEG 格式的图像输出：

```
$ go build gopl.io/ch3/mandelbrot
$ go build gopl.io/ch10/jpeg
$ ./mandelbrot | ./jpeg >mandelbrot.jpg
Input format = png
```

要注意 image/png 包的匿名导入语句，如果没有这一行语句，程序依然可以编译和运行，但是它将不能正确识别和解码 PNG 格式的图像：

```
$ go build gopl.io/ch10/jpeg
$ ./mandelbrot | ./jpeg >mandelbrot.jpg
jpeg: image: unknown format
```

上面的代码演示了它的工作机制，标准库还提供了 GIF、PNG 和 JPEG 等格式图像的解码器，用户也可以提供自己的解码器，但是为了保持程序体积较小，很多解码器并没有被全部包含，除非是明确需要支持的格式。

image.Decode 函数在解码时会依次查询支持的格式列表。每个格式驱动列表的每个入口指定了四件事情：格式的名称；一个用于描述这种图像数据开头部分模式的字符串，用于解码器检测识别；一个 Decode 函数用于完成解码图像工作；一个 DecodeConfig 函数用于解码图像的大小和颜色空间的信息。每个驱动入口是通过调用 image.RegisterFormat 函数注册的，一般是在每个格式包的 init 初始化函数中调用的，例如 image/png 包是这样注册的：

```
package png // image/png

func Decode(r io.Reader) (image.Image, error)
func DecodeConfig(r io.Reader) (image.Config, error)

func init() {
    const pngHeader = "\x89PNG\r\n\x1a\n"
    image.RegisterFormat("png", pngHeader, Decode, DecodeConfig)
}
```

最终的效果是，主程序只需要匿名导入特定图像驱动包就可以用 image.Decode 解码对应格式的图像了。

数据库包 database/sql 也是采用了类似的技术，让用户可以根据自己的需要选择导入必要的数据库驱动。例如：

```
import (
    "database/sql"
```

仅供非商业用途或交流学习使用


```
    _ "github.com/lib/pq"              // 启用对 Postgres 的支持
    _ "github.com/go-sql-driver/mysql" // 启用对 MySQL 的支持
)
db, err = sql.Open("postgres", dbname) // OK
db, err = sql.Open("mysql", dbname)    // OK
db, err = sql.Open("sqlite3", dbname)  // 返回错误: unknown driver "sqlite3"
```

7.1.3 包的使用

为了更好地理解包导入的细节，在本节中将创建一个包，这个包很简单，首先在工作空间建立一个项目 test：

```
mkdir $GOPATH/src/test
```

在 src/test 中新建一个文件如下：

```
package test
// 公开函数
func Even(i int) bool {
      return i % 2 == 0
}
// 私有函数
func odd (i int) bool {
      return i % 2 == 1
}
```

然后保存为 test.go 文件，最后使用 go build 和 go install 命令编译和安装这个包。现在我们有了一个包，接下来看如何导入刚才建立的包，应用于新的程序之中。

新建一个文件，就叫作 main.go 吧（当然也可以命名为其他名字）：

```
package main
// 下面导入了本地包 test 和官方标准包 fmt
import (
      "test"
      "fmt"
)
// 调用 test 包中的 Even 函数。访问一个包中的函数的语法是 package.Function()
func main() {
      i := 5
      fmt.Printf("Is %d even? %v\n", i, test.Even(i))
}
```

在上面的例子中，如果使用了 odd 这个函数，在编译的时候就会报错，因为在 test 包中定义了 odd 函数为私有函数，不能被外部访问。

仅供非商业用途或交流学习使用

7.1.4 Go 语言工具箱

Go 语言的工具箱集成了一系列命令集，一方面它是一个强大的包管理器，可以用于包的查询，计算包的依赖关系，从远程版本控制系统下载包等任务；另一方面，它也是一个构建系统，计算依赖、构建程序，类似 make 命令。除此之外，它还是一个测试驱动程序，不过在本章中不会讨论关于测试的内容。

1. 包文档

根据编程规范，每个包都应该有包注释，一般是在 package 前的一个注释块。对于多文件包，包注释只需要出现在任意一个文件前即可。包注释应当对包进行介绍，并提供关于包的整体信息，这会出现在 go doc 生成的关于包的页面上，并且相关的细节会一并显示。

包中每个导出的成员和包声明前都应该包含目的和用法说明的注释。下面是来自于 fmt 包的例子：

```
// Printf formats according to a format specifier and writes to standard
// output. It returns the number of bytes written and any write error
// encountered.
func Printf(format string, a ...interface) (n int, err error)
```

为了更好地管理文档（注释），Go 语言提供了两个工具。首先是 go doc 命令，该命令打印包的声明和每个成员的文档注释。

例如：

```
$ go doc test
package test // import "test"
这是一个简单的演示包，这里一般写上这个包的简单介绍。
可导出的函数、变量成员等的解释。
... ...
```

查看某个具体包成员的注释文档：

```
$ go doc test.TestFunc
```

该命令并不需要输入完整的包导入路径或正确的大小写。例如查看 encoding/json 包的 (*json.Decoder).Decode 方法：

```
$ go doc json.decode
```

第二个工具名字叫 godoc，它提供可以相互交叉引用的 HTML 页面，输出的信息会比 go doc 要多一些。godoc 有一个在线服务：https://godoc.org，它包含了成千上万的开源包

仅供非商业用途或交流学习使用

的文档检索。你也可以在自己的工作区目录运行 godoc 服务，运行下面的命令，然后在浏览器中查看 http://localhost:8000/pkg 页面：

```
$ godoc -http :8000
```

其中-analysis=type 和-analysis=pointer 命令行标志参数用于打开文档和代码中关于静态分析的结果。

2. 内部包

原则上 Go 语言的封装只有两个特性，一个是私有的不可导出的成员，另一个是公开的可导出的成员。但有些时候需要一种中间状态——对一些包公开，对一些包不可见。这就引入了一个概念——内部包，在 Go 语言中，构建工具会对导入路径包含 internal 关键字的包做特殊处理。一个 internal 包只能被和 internal 目录有同一个父目录的包所导入。

例如，net/http/internal/chunked 内部包只能被 net/http/httputil 或 net/http 包导入，但是不能被 net/url 包导入。不过 net/url 包却可以导入 net/http/httputil 包：

```
net/http
net/http/internal/chunked
net/http/httputil
net/url
```

3. 查询包

go list 命令可以查询可用包的信息，其最简单的形式，可以测试包是否在工作区并打印它的导入路径：

```
$ go list github.com/go-sql-driver/mysql
github.com/go-sql-driver/mysql
```

go list 命令的参数还可以用...表示匹配任意的包的导入路径，可以用它来列表工作区中的所有包：

```
$ go list ...
```

或者是特定子目录下的所有包：

```
$ go list gopl.io/ch3/...
gopl.io/ch3/basename1
gopl.io/ch3/basename2
```

或者是和某个主题相关的所有包：

```
$ go list ...xml...
encoding/xml
gopl.io/ch7/xmlselect
```

仅供非商业用途或交流学习使用

go list 命令还可以获取每个包完整的元数据，而不仅仅只是导入路径，这些元数据以不同格式提供给用户。其中-json 命令行参数表示用 JSON 格式打印每个包的元数据：

```
$ go list -json hash
{
    "Dir": "/home/gopher/go/src/hash",
    "ImportPath": "hash",
    "Name": "hash",
    "Doc": "Package hash provides interfaces for hash functions.",
    "Target": "/home/gopher/go/pkg/darwin_amd64/hash.a",
    "Goroot": true,
    "Standard": true,
    "Root": "/home/gopher/go",
    "GoFiles": [
        "hash.go"
    ],
    "Imports": [
        "io"
    ],
    "Deps": [
        "errors",
        "io",
        "runtime",
          "sync",
        "sync/atomic",
        "unsafe"
    ]
}
```

查询时同样可以使用 Go template 格式化输出元数据。

7.2 自定义包

在前面提到的包的使用中，曾简单地介绍了如何新建一个包并导入到项目中。标准包为大多数的程序提供了必要的基础构件。而在 Go 语言的社区，有很多成熟的包被设计、共享、重用和改进。目前互联网上已经发布了非常多的 Go 语言开源包，它们可以通过 http://godoc.org 检索。本章将演示如果使用已有的包和创建新的包。

到目前为止，所有的例子都只在单个 main 包中。对于任何包，可以将它的代码分隔到多个文件中，只要这些文件在相同的目录中。比如使用单个 main 包，由多个单独的文件组成，并且在每个文件的第一条语句为 package main。

对于大型的应用程序，可能想要将应用程序的功能按照逻辑单元划分成不同的包。同

仅供非商业用途或交流学习使用

样，创建的包会被多个相似的应用程序使用。单个应用程序使用的包和跨应用程序使用的包没有什么差别，但是，可以创建一个隐式的差别，将单个应用程序专用包放到应用程序的子目录下，而多个程序共享包可以放到 GOPATH 的 src 目录。在 GOPATH 下的每一个目录都应该包含一个 src 的目录，这是 Go 语言规定的。

程序的源代码和包都应该放在 GOPATH/src 目录下的子目录中，打包之后还可以直接将自定义的包安装到 GOROOT 目录下。

7.2.1 包的制作

创建自定义包最好是在GOPATH/src 目录下(或者 GOPATH/src 下的其中一个子目录)。单个应用程序专用包可以在应用程序的目录中创建，而分享类型的包应当直接在 GOPATH/src 目录中创建。

按照约定，一个包的源代码放在和包名称相同的目录中。我们按自己喜欢的方式，将源代码划分到多个文件中，这些文件的名字可以是任意的（只要它们以.go 结尾）。

如果在项目目录中执行 go build 命令，将获得一个可执行文件（名字为包名的可执行文件）。可是，如果想在 GOPATH/bin 目录中生成可执行文件，或者如果想让其他应用程序也可以使用这个包，必须使用 go install 安装。

当使用 go install 编译 Go 语言程序时，它将创建两个目录(如果不存在)：GOPATH/bin 创建对应可执行程序文件，GOPATH/pkg/linux_amd64/创建包含了静态的包的二进制库文件（以 Linux 64 位为例）。

安装之后的包可以通过在 Go 语言程序中使用 import "example/test"导入，这里给定的是完整的路径，除了 GOPATH/src 部分，实际上，GOPATH 下的任何程序或者包都可以使用这种导入，因为 Go 语言不会区分应用程序专用的包和共享的包。

7.2.2 特定平台的代码

在某些情况下，需要不同平台的代码。比如，在类 UNIX 系统的 shell 中可以使用通配符，所以命令行中的*.txt 可以在程序中接受，如["README.txt", "INSTALL.txt"]保存在 os.Args[1:] slice 中。但是在 Windows 系统中，程序仅获得["*.txt"]，因此可以在 Windows 上使用 filepath.Glob()函数达到同样的通配符功能。

在程序运行时决定是否使用 filepath.Glob()的解决方案是通过 runtime.GOOS==

仅供非商业用途或交流学习使用

"windows"来判断当前运行时的操作系统为 Windows。另一个解决方案是不同平台的代码放到对应的文件中，不同平台的文件可以定义名字相同但功能不同的函数。

当使用 go build 编译包时，在 Linux 的机器上会编译 Linux 平台的代码，而其他平台的文件将会被忽略。

7.2.3 godoc 生成文档

Go 语言的包，特别是对于打算分享的包，都需要一份规范的文档。Go 语言提供了一个文档工具——godoc，它可以用来在控制台显示包的文档和函数，或者作为一个 Web 服务器，使得文档以网页的形式显示。

如果这个包在 GOPATH 下，godoc 将自动找到它，并且在网页的左侧提供一个到这个包的链接。如果这个包没有在 GOPATH 下，可以运行 godoc -path 选项，将包的路径赋值给它。

本章不会讨论文档规范细节。在默认情况下，godoc 只显示导出的 types、classes、constants 以及 variables，所有这些都应该被文档化，文档是直接写在源文件中的。

对于一个包，在 package 语句之前添加整个包的描述，而第一句话是整个包的摘要：

```
// Map is a key-ordered map.
// The zero value is an invalid map! Use one of the construction functions
// (e.g., New()), to create a map for a specific key type.
type Map struct {
```

这个文档用于导出的类型，必须写在 type 语句之前，并且指明这个类型的零值是否有效：

```
// New returns an empty Map that uses the given less than function to
// compare keys. For example:
//      type Point { X, Y int }
//      pointMap := omap.New(func(a, b interface{}) bool {
//          α, β := a.(Point), b.(Point)
//          if α.X != β.X {
//              return α.X < β.X
//          }
//          return α.Y < β.Y
//      })
func New(less func(interface{}, interface{}) bool) *Map {
```

这段文档用于函数和方法，它们必须写在函数的第一条语句之前，注释中的缩进最终会渲染为 HTML 的代码。

仅供非商业用途或交流学习使用

下面展示了一个便利的构造函数的文档：

```
// Insert inserts a new key-value into the Map and returns true; or
// replaces an existing key-value pair's value if the keys are equal and
// returns false. For example:
//      inserted := myMap.Insert(key, value).
func (m *Map) Insert(key, value interface{}) (inserted bool) {
```

上面是 Insert()方法的文档。需要稍加注意的是，在 Go 语言中有一个约定是函数和方法的文档，都是以这个函数或者方法的名称开始的，并且不要加上圆括号。

7.2.4 包的打包与发布

此处的打包发布不仅指 Go 语言包项目在本地编译和安装，还包括分发到其他设备系统中的打包方法。在前面的内容中，Go 语言包的编译安装其实就是一个打包过程（go install 命令），但这个过程仅针对本地而言，面向的是开发者们的打包发布方式。对于一些企业级用户而言，应该尽量把分发部署操作简化，最理想的状态就是把整个项目打包成一个文件，拿到这个文件的人可以直接执行就完成部署了。

目前 Go 语言的确能够实现这种高度集成的发布方式，而且还不止一种方式。目前社区中比较主流的处理方式有三种，一是使用 Go.rice，还有就是使用 Statik，最后一个是比较老旧但接口丰富的 go-bindata，这三个项目的地址如下：

https://github.com/GeertJohan/go.rice

https://github.com/rakyll/statik

https://github.com/jteeuwen/go-bindata

安装这三个包的方式都是使用 go get 获取的：

```
go get github.com/rakyll/statik
go get github.com/GeertJohan/go.rice
go get github.com/GeertJohan/go.rice/rice # 此处 rice 是一个单独的工具
go get github.com/jteeuwen/go-bindata
```

1. go.rice

先说 rice 这个项目，它需要在 main.go 中导入这个包：

```
import "github.com/GeertJohan/go.rice"
```

go.rice 相比其他几个项目不同的是，它不仅仅是一个打包工具，还是一个静态文件操作库，自带文件访问服务。rice 会把一个目录认为是一个 rice.Box 来操作：

仅供非商业用途或交流学习使用


```
// 示例代码 7-2
import (
    "fmt"
    "html/template"

    "github.com/GeertJohan/go.rice"
)

func main() {
    // 这里写相对于执行文件的地址
    box, err := rice.FindBox("theme/default")
    if err != nil {
        println(err.Error())
        return
    }
    // 从目录 Box 读取文件
    str, err := box.String("post.html")
    if err != nil {
        println(err.Error())
        return
    }
    t, err := template.New("tpl").Parse(str)
    fmt.Println(t, err)
}
```

go.rice 的打包命令是 rice，用起来非常直接：在有使用 go.rice 操作的 Go 代码目录中，直接执行 rice embed-go：

```
rice embed-go
rice -i "github.com/xxx/ooo" embed-go # -i 处理指定包里的 go.rice 操作
```

它会生成当前包名下的、嵌入了文件的代码文件 rice-box.go。但是，它不递归处理 import，它会分析当前目录下的 Go 代码中 go.rice 的使用，找到对应的需要嵌入的文件夹。但是子目录下和 import 包里面的 go.rice 使用情况则不会分析，需要开发者手动进入相应文件夹执行命令，或者使用-i 指定要处理的包。这样非常不友好，所以也是 rice 的一大缺憾。

不过 rice 是直接支持 http.FileSystem 接口的，可以实现类似 Nginx 静态资源访问的页面：

```
// MustFindBox 出错直接 panic
http.Handle("/", http.FileServer(rice.MustFindBox("theme").HTTPBox()))
http.ListenAndServe(":1234", nil)
```

但是 rice.FindBox(dir)只能加载一个目录，因此如果遇到多个目录的场景，可以这样写：

仅供非商业用途或交流学习使用


```
    http.Handle("/img", http.FileServer(rice.MustFindBox("static/img").HTTPBox()))
    http.Handle("/css", http.FileServer(rice.MustFindBox("static/css").HTTPBox()))
    http.Handle("/js", http.FileServer(rice.MustFindBox("static/js").HTTPBox()))
    http.ListenAndServe(":1234", nil)
```

2. statik

根据上面命令进行安装之后即可使用，假设要把 assets 目录里的静态资源文件 demo.txt 嵌入到执行文件中，首先要导入 statik 这个包：

```
// 示例代码 7-3
import (
    "fmt"
    "io/ioutil"
    "os"

    _ "testProject/statik" // 自动生成的目录
    "github.com/rakyll/statik/fs"
)

// 开始构建之前先执行 go generate
func main() {
    statikFS, err := fs.New()
    if err != nil {
        fmt.Printf("err: %v\n", err)
        os.Exit(1)
    }
      // 导入的目录路径和上面 rice 的用法一样
    file, err := statikFS.Open("/assets/demo.txt")
    if err != nil {
        fmt.Printf("err: %v\n", err)
        os.Exit(1)
    }
      // 读取文件内容并输出
    content, err := ioutil.ReadAll(file)
    if err != nil {
        fmt.Printf("err: %v\n", err)
        os.Exit(1)
    }
    fmt.Printf("content: %s\n", content)
}
```

然后执行：statik -src=./assets && go generate，这段代码的功能就是从 statikFS 提供的文件系统接口获取/assets/demo.txt 这个文件的内容并输出，可以看到操作起来和操作普通文件的方法基本一致。

上面命令会生成一个 statik 目录，里面存放的是自动生成的 Go 文件，所有 assets 下

的文件变成一个压缩后的字符串放在这个文件中，并且在程序启动时会解析这个字符串，构造一个 http.FileSystem 对象，之后就可以使用对文件系统类似的操作来获取文件内容。

statik 同样也实现了标准库中的 http.FileSystem 接口，所以也可以直接使用 http 包提供静态资源的访问服务，关键部分代码如下：

```
import (
  "github.com/rakyll/statik/fs"
  _ "testProject/statik/statik"
)
...
  statikFS, err := fs.New()
  if err != nil {
    log.Fatal(err)
  }
...
  http.Handle("/assets/", http.StripPrefix("/assets/", http.FileServer(statikFS)))
  http.ListenAndServe(":8080", nil)
```

现在编译执行之后，可以在 http://localhost:8080/assets/中看到打包后的那些文件。

3. go-bindata

安装 go-bindata 命令如下：

```
go get github.com/jteeuwen/go-bindata/...
```

注意 go get 地址最后有三个点，这样安装时会分析所有子目录并下载依赖编译子目录内容。go-bindata 的命令工具在子目录中（上面的 rice 项目也可以使用类似的方式安装而不用输入两次 go get 命令）。

最简单的用法就是：go-bindata data/，它会在当前工作目录中生成一个 bindata.go 文件，并打包 data 目录中的所有资源文件（如果要递归子文件夹应该加上三个点：go-bindata data/...）。

```
go-bindata -o=data/asset.go -pkg=data source/... theme/... doc/src/... doc/images/...
```

上面是一个稍微复杂的用法，其中-o 表示输出文件到 data/asset.go，包名-pkg=data，然后是需要打包的目录，三个点包括所有子目录。这样就可以把所有相关文件打包到 asset.go 中，包名为 package data，保持和目录一致。

接下来是释放静态文件的代码：

```
dirs := []string{"source", "theme", "doc"} // 设置需要释放的目录
for _, dir := range dirs {
```

```
    // 解压 dir 目录到当前目录
    if err := asset.RestoreAssets("./", dir); err != nil {
        isSuccess = false
        break
    }
}
if !isSuccess {
    for _, dir := range dirs {
        os.RemoveAll(filepath.Join("./", dir))
    }
}
```

asset.go 内的静态内容根据实际的目录位置索引，所以可以直接通过目录或者文件地址去操作。

go-bindata 支持开发模式，即不嵌入静态内容，只生成操作方法到输出的 Go 代码中，在命令后面加上-debug 即可。

-debug 参数生成的代码会直接将静态文件读取到内存，而不是编码到代码中。这使得代码文件更小，你可以更快速地编写业务逻辑，等到一切正常再一次性打包，不必每次花费时间在打包上。

```
// -pkg=asset, 打包的包名是 asset
bytes, err := asset.Asset("theme/default/post.html")    // 根据地址获取对应内容
if err != nil {
    fmt.Println(err)
    return
}
t, err := template.New("tpl").Parse(string(bytes))      // 比如用于模板处理
fmt.Println(t, err)
```

使用-ignore 参数可以忽略某些文件夹，除此之外 go-bindata 的第三方包 go-bindata-assetfs 还实现了 HTTP 静态文件服务接口，支持 HTTP 访问静态文件目录。

扩展包地址：https://github.com/elazarl/go-bindata-assetfs。

以我们上面编译好的 asset.go 为例：

```
// 示例代码 7-4
import (
    "net/http"

    "github.com/elazarl/go-bindata-assetfs"
    "demo/data/asset"  // 测试项目
)

func main() {
```

```
    fs := assetfs.AssetFS{
        Asset:     asset.Asset,
        AssetDir:  asset.AssetDir,
        AssetInfo: asset.AssetInfo,
    }
    http.Handle("/", http.FileServer(&fs))
    http.ListenAndServe(":1234", nil)
}
```

访问 http://localhost:1234，就可以看到嵌入的 source、theme、doc 三个目录的列表页面，和 Nginx 查看静态文件目录是类似的。

7.2.5 自定义包的导入

自定义包的导入与 7.1.2 并无差别，不过 Go 语言允许对导入的包使用别名来标识，这使得开发者可以在不同应用场景下切换导入的包而不改变应用原有的业务代码。例如，开发中的项目有一个专门提供 API 接口的模块被打包为一个模块包，api 包拥有多个版本（例如 api_v1、api_v2 等），但在实际业务代码中，只需把 api 包导入并命名为 api（例如 import api "api_v1"），此后切换版本只需改动 api_v1 为 api_v2 即可（即为 import api "api_v2"）。

以上都基于一个前提，那便是 api_v1 和 api_v2 的 API 必须是兼容的，这样其余所有的代码都不需要做任何改动。对于导入的第三方包，不要使用标准库的包名作为别名，也别对标准库的包使用自定义的别名，避免造成不必要的误会。

之前在第 5 章提到过，当导入一个包时，它所有的 init 函数就会被执行。有些时候并非真的需要使用这些包，仅仅是希望它的 init 函数被执行而已。

例如下面例子中（Docker 日志驱动的初始化代码），导入了不少项目模块的包，但并不意味着这个文件的代码会用到这些包里面的某些函数：

```
package daemon // 其他文件导入 import "daemon"

import (
      // 在这里导入包只是为了确保它们的 init 函数被调用
      _ "daemon/logger/awslogs"
      _ "daemon/logger/fluentd"
      _ "daemon/logger/gcplogs"
      _ "daemon/logger/gelf"
      _ "daemon/logger/journald"
      _ "daemon/logger/jsonfilelog"
      _ "daemon/logger/logentries"
```

```
    _ "daemon/logger/splunk"
    _ "daemon/logger/syslog"
)
```

这个例子中导入了众多 logger 模块，这些模块内部的函数并不会被当前文件调用，只有在导入 daemon 模块的文件（import "daemon"）时才会执行这些包的 init()函数用于初始化，init()函数会为 daemon 包注册这些导入的日志引擎。由于没有用到这些包提供的功能，所以可以把这些包的别名都写为空白标识符。

7.3 本章小结

在 Go 语言中包是组织代码的基本单位。环境变量 GOPATH 决定了 Go 语言源代码在磁盘上被保存、编译和安装的位置，可以为每个工程设置不同的 GOPATH，以保持源代码和依赖的隔离。

Go 语言工具是最常用的开发工具，开发人员可以使用 go get 来获取别人的包并将其安装到自己的 GOPATH 指定的目录。想要为别人创建包很简单，只要把源代码放到公用代码库，并遵守一些简单的规则就可以了，Go 语言在设计时将分享代码作为语言的核心特性和驱动力。

推荐使用依赖管理工具来管理依赖，有很多社区开发的依赖管理工具，如 godep、vender 和 gb 都是很不错的包管理工具。

第 8 章
结构体与方法

前面已经多次使用过自定义类型，这种自定义类型正是通过 Go 语言类型别名（alias types）和结构体的形式支撑的。结构体也是复合类型，通常用一个带属性的结构体来表示一个现实中的实体。

一个自定义类型由一系列属性组成，每个属性有自己的类型和值。结构体的目的就是把数据聚集在一起，以便能够更加便捷地操作这些数据，在外部来看就像是一个实体。记住，结构体依旧是值类型，因此可以通过 new 函数来创建。

8.1 结构体

结构体的概念在 C 语系里很常见，基本都称为 struct，Go 语言也是如此。在面向对象的编程语言中，类似一个无方法的轻量级类。不过前面说过，Go 语言中没有类的概念，因此在 Go 语言中结构体有着更为特别且重要的地位。

8.1.1 结构体定义

结构体定义的一般方式如下，组成结构体类型的那些数据称为字段（fields）。每个字段都有一个类型和一个名字；在一个结构体中，字段名字必须是唯一的：

```
type author struct {
    field1 type1
    field2 type2
    ...
}
```

此外，type T struct {a, b int}也是合法语法，它更适用于那些简单的结构体。结构体里的字段都有名字，像 field1、field2 等，如果字段在代码中从来不会被用到，那可以命名为_（空标识符）。

结构体的字段可以是任意的类型，包括结构体本身，甚至可以是函数或者接口。声明结构体类型的一个变量，并给它的字段赋值：

```
var s T
s.a = 5
s.b = 8
```

数组可以看作一种结构体类型，不过它使用下标而不是具名字段。

1. 使用 new 创建结构体

一般在结构体定义之后，习惯使用 t := new(T)给该结构体变量分配内存，它返回指向已分配内存的指针。变量 t 是一个指向 T 的指针，此时结构体字段的值是它们所属类型的零值。注意定义和分配内存两部分代码一定要放在一起，例如结构体定义范围是整个包，但是内存可以选择在合适的时候分配。

```
var t *T
t = new(T)
```

如果声明 var t T，那么也会给 t 分配内存，并初始化为零值，这个时候 t 是类型 T 的。这两种方式中，t 被称作类型 T 的一个实例（instance）或对象（object）。

下面是一个非常简单的例子：

```
// 示例代码 8-1
package main
import "fmt"

type myStruct struct {
    i1  int
    f1  float32
    str string
}

func main() {
    ms := new(myStruct)
    ms.i1 = 10
    ms.f1 = 15.5
    ms.str= "Google"

    fmt.Printf("int: %d\n", ms.i1)
```

```
    fmt.Printf("float: %f\n", ms.f1)
    fmt.Printf("string: %s\n", ms.str)
    fmt.Println(ms)
}
/*
返回:
int: 10
float: 15.500000
string: Google
&{10 15.5 Google}
*/
```

fmt.Println 可以打印结构体的属性值，像在面向对象语言中那样，使用点符号给字段赋值：

```
structname.fieldname = value
```

同理，点符号也可以用于获取结构体字段的值：

```
structname.fieldname
```

这个点符号在 Go 语言中叫作选择器（selector），无论变量是一个结构体类型还是一个结构体类型指针，都可以使用相同的选择器符号（selector-notation）来引用结构体的字段：

```
type myStruct struct { i int }
var v myStruct    // v是结构体类型变量
var p *myStruct   // p是指向一个结构体类型变量的指针
v.i
p.i
```

初始化一个结构体也可以通过一个结构体字面量（struct-literal）来实现（这样更简短）：

```
ms := &myStruct{10, 15.5, "Google"}
// 此时 ms 的类型是 *struct1
// 或者
var ms myStruct
ms = myStruct{10, 15.5, "Google"}
```

我们把上面第一种写法称为混合字面量语法（composite literal syntax）：&T{....}这种写法依旧会调用 new()初始化，而且大括号里面的值必须按照顺序填写。

在下面的例子中，通过在值的前面放上字段名来初始化字段，显然，表达式 new(Type)和&Type{}是等价的：

```
type Sleep struct {
    start int
    end   int
}
```

```
// 三种初始化方式
intr := Sleep{0, 3}
intr := Sleep{end:5, start:1}
intr := Sleep{end:5}
```

在以上的初始化方式中，第一种的值必须以字段在结构体定义时的顺序填写，这里的符号&可以省略，毕竟不是直接通过字面量初始化的，前面已经定义了。第二种方式是字段名加一个冒号放在值的前面，在这种情况下值的顺序不必一致，并且某些字段还可以被忽略掉，就像初始化代码示例中的第三行那样。

结构体类型和字段的命名遵循可见性规则，一个导出的结构体类型中可以只有一些字段是导出的，另一些不可以导出。类型在定义它的包中必须是唯一的，它的完全类型命名格式就是常见的 pack.struct 的形式。

下面的例子中有一个结构体 Person 和一个方法，方法有一个类型为*Person 的参数（所以对象本身是可以被改变的），下面使用三种不同的方式调用这个方法：

```
// 示例代码 8-2
package main
import (
    "fmt"
    "strings"
)

type Person struct {
    firstName   string
    lastName    string
}

func upPerson(p *Person) {
    p.firstName = strings.ToUpper(p.firstName)
    p.lastName = strings.ToUpper(p.lastName)
}

func main() {
    // 作为值类型
    var pers1 Person
    pers1.firstName = "张"
    pers1.lastName = "三三"
    upPerson(&pers1)
    fmt.Printf("这个人的名字是 %s %s\n", pers1.firstName, pers1.lastName)

    // 作为指针
    pers2 := new(Person)
    pers2.firstName = "张"
```

```
    pers2.lastName = "三三"
    (*pers2).lastName = "三三"  // 这也是合法的
    upPerson(pers2)
    fmt.Printf("这个人的名字是 %s %s\n", pers2.firstName, pers2.lastName)

    // 作为字面量
    pers3 := &Person{"张","三三"}
    upPerson(pers3)
    fmt.Printf("这个人的名字是 %s %s\n", pers3.firstName, pers3.lastName)
}
/*
返回:
这个人的名字是 张 三三
这个人的名字是 张 三三
这个人的名字是 张 三三
*/
```

在第二种情况中，Go 语言可以直接通过指针（pers2.lastName）给结构体字段赋值，不必像 C++中那样需要使用->操作符，Go 语言会自动转换。也可以通过解析指针来设置值，例如上面(*pers2).lastName 这样。

在一些语言中，结构体是引用类型，一个对象和它里面包含的对象可能会在不同的内存空间中。但 Go 语言不一样，Go 语言的结构体与它的数据在内存中是以连续内存块存在的（即便是嵌套结构体也是一样的），这样可以带来不小的性能上的优势。

2. 递归结构体

结构体类型可以通过引用自身来定义，这在定义链表或二叉树的元素（通常叫节点）时特别有用，此时节点包含指向临近节点的链接（地址）。

如下所示，data 字段用于存放有效数据（比如 float64），su 指针指向后继节点：

```
type Node struct {
    data    float64
    su      *Node
}
```

链表中的第一个元素叫作 head，它指向第二个元素；最后一个元素叫作 tail，它没有后继元素，所以它的 su 值为 nil。当然真实的链接会有很多数据节点，并且链表可以动态增长或收缩。

同样可以定义一个双向链表，它有一个前趋节点 pr 和一个后继节点 su：

```
type Node struct {
    pr      *Node
    data    float64
```

```
    su      *Node
}
```

二叉树中每个节点最多能链接至两个节点：左节点（le）和右节点（ri），这两个节点本身又可以有左右节点，以此类推。树的顶层节点叫作根节点（root），底层没有子节点的节点叫作叶子节点（leaves），叶子节点的 le 和 ri 指针为 nil 值。在 Go 语言中可以如下定义二叉树：

```
type Tree strcut {
    le      *Tree
    data    float64
    ri      *Tree
}
```

3. 结构体转换

Go 语言类型转换规则非常严格。当给结构体定义一个别名（alias）类型时，该结构体类型与别名（alias）类型的底层类型都是一样的，可以直接转换，不过需要注意其中由非法赋值或转换引起的编译错误：

```
// 示例代码 8-3
package main
import "fmt"

type number struct {
    f float32
}

type nr number   // 类型别名

func main() {
    a := number{5.0}
    b := nr{5.0}

      // var i float32 = b
      // 编译错误返回：
      // compile-error: cannot use b (type nr) as type float32 in assignment
      // var i = float32(b)
      // 编译错误返回：
      // compile-error: cannot convert b (type nr) to type float32
      // var c number = b
      // compile-error: cannot use b (type nr) as type number in assignment
      // 此处需要转换
    var c = number(b)
    fmt.Println(a, b, c)
}
```

```
/*
返回:
{5} {5} {5}
*/
```

8.1.2 结构体使用

1. 自定义包的结构体

在下面的例子中，main.go 使用了一个结构体，它来自 struct_pack 下的包 structPack：

```
package structPack

type ExpStruct struct {
    Mi1 int
    Mf1 float32
}
```

下面是 main.go 的代码：

```
// 示例代码 8-4
package main
import (
    "fmt"
    "./struct_pack/structPack"
)

func main() {
    struct1 := new(structPack.ExpStruct)
    struct1.Mi1 = 10
    struct1.Mf1 = 16.

    fmt.Printf("Mi1 = %d\n", struct1.Mi1)
    fmt.Printf("Mf1 = %f\n", struct1.Mf1)
}
```

编译执行输出：

```
Mi1 = 10
Mf1 = 16.000000
```

2. 结构体成员访问

在 Go 语言中，访问结构体成员需要使用点号操作符，格式为“结构体.成员名”。

```
// 示例代码 8-5
package main

import "fmt"
```

```
type Employee struct {
    ID          int
    Name      string
    Address      string
    Phone        string
}

func main() {
   var employee1 Employee

   /* employee1 描述 */
   employee1.ID = 10001
   employee1.Name = "Tom"
   employee1.Address = "xxxx"
   employee1.Phone = "1881412xxxx"

   /* 打印 employee1 信息 */
   fmt.Printf( "employee1 ID : %d\n", employee1.ID)
   fmt.Printf( "employee1 Name : %s\n", employee1.Name)
   fmt.Printf( "employee1 Address : %s\n", employee1.Address)
   fmt.Printf( "employee1 Phone : %s\n", employee1.Phone)
}
/*
返回:
employee1 ID : 10001
employee1 Name : Tom
employee1 Address : xxxx
employee1 Phone : 1881412xxxx
*/
```

可以定义指向结构体的指针（指向结构体变量的首地址），类似于其他指针变量，格式如下：

```
var employee *Employee
```

使用结构体指针访问结构体成员，使用“.”操作符：

```
struct_pointer.title;
```

接下来使用结构体指针重写以上实例，代码如下：

```
// 示例代码 8-6
package main

import "fmt"

type Employee struct {
    ID          int
```

```
    Name      string
    Address       string
    Phone         string
}

func main() {
   var employee Employee

   employee.ID = 10001
   employee.Name = "Tom"
   employee.Address = "xxxx"
   employee.Phone = "1881412xxxx"

   /* 打印 employee 信息 */
   printEmployee(&employee)
}
func printEmployee( employee *Employee ) {
   fmt.Printf( "Employee ID : %d\n", employee.ID)
   fmt.Printf( "Employee Name : %s\n", employee.Name)
   fmt.Printf( "Employee Address : %s\n", employee.Address)
   fmt.Printf( "Employee Phone : %s\n", employee.Phone)
}
/*
返回:
Employee ID : 10001
Employee Name : Tom
Employee Address : xxxx
Employee Phone : 1881412xxxx
*/
```

3. 结构体参数传输

我们可以像其他数据类型一样将结构体类型作为参数传递给函数，并以以上实例的方式访问结构体变量，有形式参数传输和指针参数传输两种形式，举例如下：

```
// 示例代码 8-7
package main
import "fmt"

type Employee struct {
    ID           int
    Name      string
    Address       string
    Phone         string
}

func main() {
   var employee Employee
```

```
    employee.ID = 10001
    employee.Name = "Tom"
    employee.Address = "xxxx"
    employee.Phone = "18814128405"

    fmt.Printf( "形式传参之前, employee ID : %d\n", employee.ID)
    operateEmployee1(employee)
    fmt.Printf( "形式传参之后, employee ID : %d\n", employee.ID)

    fmt.Printf( "指针传参之前, employee ID : %d\n", employee.ID)
    operateEmployee2(&employee)
    fmt.Printf( "指针传参之后, employee ID : %d\n", employee.ID)
}
//形式传参
func operateEmployee1( employee Employee) {
    employee.ID = 10010
}

//指针传参
func operateEmployee2( employee *Employee) {
    employee.ID = 10010
}
/*
返回:
形式传参之前, employee ID : 10001
形式传参之后, employee ID : 10001
指针传参之前, employee ID : 10001
指针传参之后, employee ID : 10010
*/
```

由结果可以看出，形式传参中 employee 只是传递了一个副本到另一个函数中，函数中操作的是副本，对 employee 没有任何影响；而在指针传参中 employee 传递的是地址，函数中的操作会影响到 employee。

8.1.3　带标签的结构体

结构体中的字段不仅有名字与类型，还有一个可选的附属于字段的标签（tag）：用于文档说明或者开发过程中的特殊标记，它的值是一串字符串。标签内容只有 reflect 包能获取。

```
// 示例代码 8-8
package main

import (
    "fmt"
    "reflect"
```

仅供非商业用途或交流学习使用


```
)

type TagType struct { // tags
    field1 bool   "是否有存货"
    field2 string "商品名称"
    field3 int    "商品价格"
}

func main() {
    tt := TagType{true, "LPhone X", 1}
    for i := 0; i < 3; i++ {
        refTag(tt, i)
    }
}
/*
在一个变量上调用 reflect.TypeOf()可以获取变量的正确类型
如果变量是一个结构体类型，就可以通过 Field 来索引结构体的字段，然后就可以使用 Tag 属性
*/
func refTag(tt TagType, ix int) {
    ttType := reflect.TypeOf(tt)
    ixField := ttType.Field(ix)
    fmt.Printf("%v\n", ixField.Tag)
}
/*
返回:
是否有存货
商品名称
商品价格
*/
```

8.1.4 匿名字段和内嵌结构体

结构体可以包含一个或多个匿名（或内嵌）字段，即这些字段没有显式的名字，只有字段的类型是必需的，此时类型就是字段的名字。匿名字段本身可以是一个结构体类型，即结构体可以包含内嵌结构体。

1. 匿名字段

匿名字段和面向对象编程中的继承概念相似，可以被用来模拟类似继承的行为。Go 语言中的继承是通过内嵌或组合来实现的，所以可以说在 Go 语言中，组合比继承更受欢迎。

```
// 示例代码 8-9
package main
```

仅供非商业用途或交流学习使用


```
import "fmt"

type firstS struct {
    in1 int
    in2 int
}

type secondS struct {
    b    int
    c    float32
    int  // 匿名字段
    firstS // 匿名字段
}

func main() {
    sec := new(secondS)
    sec.b = 6
    sec.c = 7.5
    sec.int = 60
    sec.in1 = 5
    sec.in2 = 10

    fmt.Printf("sec.b is: %d\n", outer.b)
    fmt.Printf("sec.c is: %f\n", outer.c)
    fmt.Printf("sec.int is: %d\n", outer.int)
    fmt.Printf("sec.in1 is: %d\n", outer.in1)
    fmt.Printf("sec.in2 is: %d\n", outer.in2)

    // 使用结构体字面量
    sec2 := secondS{6, 7.5, 60, firstS{5, 10}}
    fmt.Println("sec2 is:", sec2)
}
/*
返回:
sec.b is: 6
sec.c is: 7.500000
sec.int is: 60
sec.in1 is: 5
sec.in2 is: 10
sec2 is:{6 7.5 60 {5 10}}
*/
```

通过类型 sec.int 的名字来获取存储在匿名字段中的数据，于是可以得出一个结论：在一个结构体中对于每一种数据类型只能有一个匿名字段。

2. 内嵌结构体

结构体也是一种数据类型，所以它同样可以作为匿名字段使用，上面的例子中就是这

仅供非商业用途或交流学习使用

样使用的。外层结构体通过 sec.in1 直接进入内层结构体的字段，内嵌结构体甚至可以来自其他包。内层结构体被简单地插入或者内嵌进外层结构体，这种简单的“继承”机制使得 Go 语言很轻松就能实现从一个或一些类型中继承部分或全部的实现。

```
// 示例代码 8-10
package main

import "fmt"

type A struct {
    ax, ay int
}

type B struct {
    A
    bx, by float32
}

func main() {
    b := B{A{1, 2}, 3.0, 4.0}
    fmt.Println(b.ax, b.ay, b.bx, b.by)
    fmt.Println(b.A)
}
/*
返回:
1 2 3 4
{1 2}
*/
```

3. 命名冲突

当两个字段拥有相同的名字（例如继承其他结构体）时会产生冲突，一般情况有两种：

（1）外层字段的名字覆盖内层字段的名字，但是两者的内存空间都会保留，这提供了一种重载字段或方法的方式。

（2）相同的名字在同层次结构体中出现了重复，并且这个字段被程序调用了，这将导致程序错误（字段不被调用不会报错，但有隐患）。这种情况只能由程序员自己修正。

看下面例子：

```
type A struct {a int}
type B struct {a, b int}
```

仅供非商业用途或交流学习使用


```
type C struct {A; B}
var c C;
```

后续代码中将不能使用 c.a 这种方式，因为编译器分不清是 c.A.a 还是 c.B.a，因此会报错。

```
type D struct {B; b float32}
var d D;
```

后续代码中使用 d.b 没问题，因为它的类型是 float32，而不是 B 的 b 类型。如果想要调用内层的 b，可以通过 d.B.b 得到对应的值。

8.2　类型系统

顾名思义，类型系统是指一个语言的类型体系结构，类型系统描述的是这些内容在一个语言中如何被关联。Go 语言是一种静态类型的编程语言，这意味着，编译器需要在编译时知晓程序里每个值的类型。如果提前知道类型信息，编译器就可以确保程序合理地使用值，这有助于减少潜在的内存异常和 bug，并且使编译器有机会对代码进行一些性能优化，提高执行效率。

值的类型给编译器提供两部分信息：第一部分，需要分配多少内存给这个值（即值的规模）；第二部分，这段内存表示什么。对于许多内置类型来说，规模和表示是类型名的一部分。例如 int64 类型的值需要 8B（64b），表示一个整数值；float32 类型的值需要 4B（32b），表示一个 IEEE-754 定义的二进制浮点数；bool 类型的值需要 1B（8b），表示布尔值 true 或 false。

有些类型的内部表示与编译代码的机器的体系结构有关。例如，根据编译所在的机器的体系结构，一个 int 值的大小可能是 8B（64b），也可能是 4B（32b）。还有一些与体系结构相关的类型，如 Go 语言里的所有引用类型。好在创建和使用这些类型的值的时候，不需要了解这些与体系结构相关的信息。但是，如果编译器不知道这些信息，就无法阻止用户做一些导致程序受损甚至机器故障的事情。

8.2.1　用户自定义类型

Go 语言允许用户定义类型。当用户声明一个新类型时，这个声明就给编译器提供了一个框架，告知必要的内存大小和表示信息。声明后的类型与内置类型的运作方式类似。

仅供非商业用途或交流学习使用

Go 语言里声明用户定义的类型有两种方法。最常用的方法是使用关键字 struct，它可以让用户创建一个结构类型。

结构类型通过组合一系列固定且唯一的字段来声明，如下面代码所示。结构里每个字段都会用一个已知类型声明，这个已知类型可以是内置类型，也可以是其他用户定义的类型。

```
// user 在程序里定义一个用户类型
type user struct {
      name       string
      email      string
      ext        int
      privileged bool
}
```

在以上代码中，可以看到一个结构类型的声明，这个声明以关键字 type 开始，之后是新类型的名字，最后是关键字 struct。这个结构类型有 4 个字段，每个字段都基于一个内置类型。读者可以看到这些字段是如何组合成一个数据结构的。一旦声明了类型就可以使用这个类型创建值。

下面代码展示了如何声明一个 user 类型的变量，并使用某个非零值作为初始值。首先给出一个变量名，之后是短变量声明操作符。这个操作符是冒号加一个等号（:=）。一个短变量声明操作符在一次操作中会完成两件事情：声明一个变量，并初始化。短变量声明操作符会使用右侧给出的类型信息作为声明变量的类型。

```
// 声明 user 类型的变量，并初始化所有字段
lisa := user{
      name: "Tom",
      email: "tom@example.com",
      ext: 123,
      privileged: true,
}
```

既然要创建并初始化一个结构类型，就使用结构字面量来完成这个初始化，如下面代码所示。结构字面量使用一对大括号括住内部字段的初始值。

```
user{
      name: "Tom",
      email: "tom@example.com",
      ext: 123,
      privileged: true,
}
```

结构字面量可以对结构类型采用两种形式。上面代码中使用了第一种形式，这种形式

仅供非商业用途或交流学习使用

在不同行声明每个字段的名字及对应的值，字段名与值用冒号分隔，每一行以逗号结尾，这种形式对字段的声明顺序没有要求；第二种形式没有字段名，只声明对应的值，如下面代码所示：

```
lisa := user{"Tom", "tom@example.com", 123, true}
```

每个值也可以分别占一行，不过习惯上这种形式会写在一行里，结尾不需要逗号。在这种形式下，值的顺序很重要，必须要和结构声明中字段的顺序一致。当声明结构类型时，字段的类型并不限制在内置类型，也可以使用其他用户定义的类型，如下面代码所示：

```
type admin struct {
      person user
      level string
}
```

这里展示了一个名为 admin 的新结构类型。这个结构类型有一个名为 person 的 user 类型的字段，还声明了一个名为 level 的 string 字段。当创建具有 person 这种字段的结构类型的变量时，初始化用的结构字面量会有一些变化，如下面代码所示：

```
 // 声明 admin 类型的变量
fred := admin{
    person: user{  // A
        name:
        "Tom",
        email:
        "tom@example.com",
        ext:
        123,
        privileged: true,
    },
    level: "super",
}
```

为了初始化 person 字段，需要创建一个 user 类型的值，上面代码 A 处就是在创建这个值，这行代码使用结构字面量的形式创建了一个 user 类型的值，并赋给了 person 字段。

另一种声明用户定义类型的方法是，基于一个已有的类型，将其作为新类型的类型说明。当需要一个可以用已有类型表示的新类型的时候，这种方法会非常好用。

标准库使用这种声明类型的方法，从内置类型创建出很多更加明确的类型，并赋予更高级的功能：

```
type Duration int64
```

上面代码是标准库的 time 包里的一个类型声明。Duration 是一种描述时间间隔的类

型，单位是纳秒（ns）。这个类型使用内置的 int64 类型作为其表示，在 Duration 类型的声明中，我们把 int64 类型叫作 Duration 的基础类型。不过，虽然 int64 是基础类型，但 Go 语言并不认为 Duration 和 int64 是同一种类型，这两个类型是完全不同的、有区别的类型。

类型 int64 的值不能作为类型 Duration 的值来用。换句话说，虽然 int64 类型是基础类型，Duration 类型依然是一个独立的类型。两种不同类型的值即便互相兼容，也不能互相赋值，编译器不会对不同类型的值做隐式转换。

8.2.2 值语义和引用语义

值语义和引用语义的差别在于赋值，比如下面的例子：

```
b = a
b.Modify()
```

如果 b 的修改不会影响 a 的值，那么此类型属于值类型。如果会影响 a 的值，那么此类型是引用类型。Go 语言中的大多数类型都基于值语义，例如基本类型（如 byte、int、bool、float32、float64 和 string 等）和复合类型（如数组、结构体和指针等）。

Go 语言中类型的值语义表现得非常彻底，之所以这么说，是因为数组。如果读者之前学过 C 语言，就会知道 C 语言中的数组比较特别，通过函数传递一个数组的时候基于引用语义，但是在结构体中定义数组变量的时候基于值语义（表现在为结构体赋值的时候，该数组会被完整复制）。

Go 语言中的数组和基本类型没有区别，是很纯粹的值类型，例如：

```
var a = [3]int{1, 2, 3}
var b = a
b[1]++
fmt.Println(a, b)
/*
返回：
[1 2 3] [1 3 3]。
*/
```

这表明 b=a 赋值语句是数组内容的完整复制。要想表达引用，需要使用指针：

```
var a = [3]int{1, 2, 3}
var b = &a
b[1]++
fmt.Println(a, *b)
/*
返回：
```

```
[1 3 3] [1 3 3]
*/
```

这表明 b=&a 赋值语句是数组内容的引用。变量 b 的类型不是[3]int，而是*[3]int 类型。Go 语言中有 4 个类型比较特别，看起来像引用类型，如下所示。

- 数组切片：指向数组（array）的一个区间。
- map：极其常见的数据结构，提供键值查询能力。
- channel：执行体（goroutine）间的通信设施。
- 接口（interface）：对一组满足某个契约的类型的抽象。

但是这并不影响我们将 Go 语言类型看作值语义。下面来看看这 4 个类型。数组切片本质上是一个区间，可以大致将[]T 表示为：

```
type slice struct {
      first *T
      len int
      cap int
}
```

因为数组切片内部是指向数组的指针，所以可以改变所指向的数组元素并不奇怪。数组切片类型本身的赋值仍然是值语义。

map 本质上是一个字典指针，可以大致将 map[K]V 表示为：

```
type Map_K_V struct {
      // ...
}
type map[K]V struct {
      impl *Map_K_V
}
```

基于指针，完全可以自定义一个引用类型，如：

```
type IntegerRef struct {
      impl *int
}
```

channel 和 map 类似，本质上是一个指针。将它们设计为引用类型而不是统一的值类型的原因是，完整复制一个 channel 或 map 并不是常规需求。同样，接口具备引用语义，是因为内部维持了两个指针，示意为：

```
type interface struct {
      data *void
      itab *Itab
}
```

接口在 Go 语言中的地位非常重要，关于接口的内部实现细节，在后面的高阶话题中再细细剖析。

8.3 方法

在 Go 语言中，Go 方法是作用在接收者（receiver）上的一个函数，接收者是某种类型的变量。所以在 Go 语言里方法是一种特殊类型的函数。

接收者几乎可以是任意类型（接口、指针除外），包括结构体类型、函数类型，可以是 int、bool、string 或数组别名类型。接收者不能是一个接口类型，因为接口是一个抽象定义，但是方法却必须是具体的实现。

接收者也不能是一个指针类型，但需要注意的是它可以是任何其他允许类型的指针。

一个类型加上它的方法就像面向对象中的一个类，不同的是，在 Go 语言中，类型的代码和与它相关的方法代码可以存在于不同的源文件中，当然它们必须在同一个包中。

8.3.1 方法声明

方法是函数，所以不允许方法重载，对于一个类型只能有一个给定名称的方法。但如果基于接收者类型，则允许重载，具有同样名字的方法可以在多个不同的接收者类型上存在，比如在同一个包里这样做是允许的：

```
func (a *denseMatrix) Add(b Matrix) Matrix
func (a *sparseMatrix) Add(b Matrix) Matrix
```

别名类型不能有它原始类型上已经定义过的方法。

1. 方法的声明

定义方法的一般格式如下：

```
func (recv receiver_type) methodName(parameter_list) (return_value_list) { ... }
```

在方法名之前，func 关键字之后的括号中指定接收者。

如果 recv 是接收者的一个实例，Method1 是接收者类型的一个方法名，那么方法调用遵循传统的选择器符号：recv.Method1()。

如果 recv 是一个指针，Go 语言会自动解析该引用值，如果调用的方法不需要使用 recv 的值，可以用_符号替换它，比如：

```
func (_ receiver_type) methodName(parameter_list) (return_value_list) { ... }
```

recv 与面向对象语言中的 this 或 self 等类似，但 recv 并不是一个关键字，Go 语言中也没有 this 和 self 这两个关键字，所以，也可以使用 this 或 self 作为接收者的实例化名字。下面是一个结构体上的简单方法的例子：

```
// 示例代码 8-11
package main

import "fmt"

type TwoInts struct {
    a int
    b int
}

func main() {
    two1 := new(TwoInts)
    two1.a = 12
    two1.b = 10

    fmt.Printf("和为：%d\n", two1.AddThem())
    fmt.Printf("将它们添加到参数：%d\n", two1.AddToParam(20))

    two2 := TwoInts{3, 4}
    fmt.Printf("和为：%d\n", two2.AddThem())
}

func (tn *TwoInts) AddThem() int {
    return tn.a + tn.b
}

func (tn *TwoInts) AddToParam(param int) int {
    return tn.a + tn.b + param
}
/*
返回：
和为：22
将它们添加到参数：42
和为：7
*/
```

下面是非结构体类型方法的例子：

```
// 示例代码 8-12
package main

import "fmt"
```

```
type IntVector []int

func (v IntVector) Sum() (s int) {
    for _, x := range v {
        s += x
    }
    return
}

func main() {
    fmt.Println(IntVector{1, 2, 3}.Sum()) // 输出是 6
}
```

类型与在它上面定义的方法必须在同一个包里定义，像 int、float 或类似的类型属于 Go 语言的基本类型，开发者无法在基本类型上定义方法，除非修改 Go 语言的源代码。

比如想在 time.Time 上定义如下方法：

```
func (t time.Time) first3Chars() string {
    return time.LocalTime().String()[0:3]
}
```

如果类型在非本地包里定义，在非本地包上面定义的方法都会得到以下错误：

```
cannot define new methods on non-local type int
```

不过别名类型并不受这个规则限制，可以把非本地包定义的类型指定一个别名，然后在别名类型上定义方法。或者像下面这样将它作为匿名类型嵌入在一个新的结构体中。这种方式定义的方法只在这个别名类型上有效：

```
// 示例代码 8-13
package main

import (
    "fmt"
    "time"
)

type myTime struct {
    time.Time // 匿名字段
}

func (t myTime) first3Chars() string {
    return t.Time.String()[0:3]
}
func main() {
    m := myTime{time.Now()}
```

```
    // 调用匿名 Time 上的 String 方法
    fmt.Println("完整的时间格式：", m.String())
    // 调用 myTime.first3Chars
    fmt.Println("前三个字符：", m.first3Chars())
}

/*
返回：
完整的时间格式：Mon Oct 24 15:34:54 Romance Daylight Time 2017
前三个字符：Mon
*/
```

2. 函数和方法的区别

函数将变量作为参数：Function1(recv)；方法在变量上被调用：recv.Method1()。

当接收者是指针时，方法可以改变接收者的值或状态，这一点函数也可以做到（当参数作为指针传递，即通过引用调用时，函数也可以改变参数的状态）。

接收者必须有一个显式的名字，这个名字必须在方法中被使用。receiver_type 叫作“接收者类型”，这个类型必须在和方法同样的包中被声明。

在 Go 语言中，“接收者类型”对应的方法不应该写在类型结构中，就像面向对象语言的类那样，降低耦合性，类型和方法之间的关联由接收者来建立。方法与结构体没有混在一起，独立之后更容易维护，使他人更容易理解。

8.3.2　为类型添加方法

Go 语言中的大多数类型都是值语义，并且都可以包含对应的操作方法。在需要的时候，你可以给任何类型（包括内置类型）“增加”新方法。而在实现某个接口时，无须从该接口继承（事实上，Go 语言根本就不支持面向对象思想中的继承语法），只需要实现该接口要求的所有方法即可。任何类型都可以被 Any 类型引用，Any 类型就是空接口，即 interface{}。例如：

```
type Integer int
func (a Integer) Less(b Integer) bool {
      return a < b
}
```

在这个例子中，我们定义了一个新类型 Integer，它和 int 没有本质上的不同，只是它为内置的 int 类型增加了一个新方法 Less()。这样实现了 Integer 后，就可以让整型像一个普通的类一样使用：

```
func main() {
    var a Integer = 1
    if a.Less(2) {
      fmt.Println(a, "Less 2")
      }
}
```

上面的这个 Integer 例子如果不使用 Go 语言的面向对象特性，而使用之前介绍的面向过程方式实现，相应的实现细节如下所示：

```
type Integer int
func Integer_Less(a Integer, b Integer) bool {
      return a < b
}
func main() {
  var a Integer = 1
  if Integer_Less(a, 2) {
      fmt.Println(a, "Less 2")
  }
}
```

在 Go 语言中，面向对象的神秘面纱被剥得一干二净。对比下面的两段代码：

```
func (a Integer) Less(b Integer) bool { // 面向对象
      return a < b
}
func Integer_Less(a Integer, b Integer) bool { // 面向过程
      return a < b
}
a.Less(2) // 面向对象的用法
Integer_Less(a, 2) // 面向过程的用法
```

可以看出，面向对象只是换了一种语法形式来表达。Go 语言中的面向对象最为直观，也无须支付额外的成本。如果要求对象必须以指针传递，有时会有额外成本，因为对象有时很小（比如 4B），用指针传递并不划算。只有在你需要修改对象的时候，才必须用指针。

8.3.3 工厂方法创建结构体

在面向对象编程中，可以通过构造子方法实现工厂模式（一般是 new Object 等），但在 Go 语言中并不能如此构造子方法，而是相应地提供了其他方案。以结构体为例，通常会为结构体类型定义一个工厂，按惯例，工厂的名字以 new 或 New 开头。假设定义了如下的 File 结构体类型：

```
// 不强制使用构造函数，首字母大写
type File struct {
```

```
    fd      int     // 文件描述符
    name    string  // 文件名
}
```

下面是这个结构体类型对应的工厂方法，它返回一个指向结构体实例的指针：

```
func NewFile(fd int, name string) *File {
    if fd < 0 {
        return nil
    }

    return &File{fd, name}
}
```

然后就可以这样调用它：

```
f := NewFile(10, "./test.txt")
```

在 Go 语言中常常像上面这样在工厂方法里使用初始化来简便地实现构造函数。

如果 File 是一个结构体类型，那么表达式 new(File)和&File{}是等价的，这可以和大多数面向对象编程语言中笨拙的初始化方式做个比较：File f = new File(...)。

可以说工厂实例化了类型的一个对象，就像在基于类的面向对象语言中那样。如果想知道结构体类型 T 的一个实例占用了多少内存，可以使用：size := unsafe.Sizeof(T{})。

强制使用工厂方法，通过应用可见性规则就可以禁止使用 new 函数，强制用户使用工厂方法，从而使类型变成私有的，就像在面向对象语言中那样：

```
// 强制使用构造函数，首字母小写
type file1 struct {
    fd       int
    filename string
}

func NewFile1(fd int, name string) *file1 {
    if fd < 0 {
        return nil
    }
    return &file1{fd, name}
}
```

在其他包里使用工厂方法时不能通过 new 的方式创建，因为 file1 是私有的，实例化该结构体的唯一方式只能通过 NewFile1 实现。

8.3.4 基于指针对象的方法

现在我们都知道了 Go 语言不支持类似其他面向对象语言那样的传统类，相反，Go 语言使用结构体替代。由于 Go 语言支持在结构体类型上定义方法，所以这一点非常类似于其他面向对象语言中的传统类方法。

在结构体类型上可以定义两种方法，分别基于指针接收器和基于值接收器。显然，从之前的内容中得知，值接收器意味着复制整个值到内存中，内存开销非常大，而基于指针的接收器仅仅需要一个指针大小的内存。

因此，性能决定了哪种方法更值得推崇，recv 最常见的是一个指向 receiver_type 的指针（因为不需要复制整个实例，但若是按值调用就会复制整个实例），特别是在接收者类型是结构体时，性能优势就更突出了。

如果想要方法改变接收者的数据，就在接收者的指针类型上定义该方法。否则，就在普通的值类型上定义方法：

```
// 示例代码 8-14
package main

import (
    "fmt"
)

type HttpResponse struct { status_code int }

func (r *HttpResponse) validResponse() { r. status_code = 200 }

func (r HttpResponse) updateStatus() string { return fmt.Sprint(r) }

func main() {
    var r1 HttpResponse // r1 是值
    r1.validResponse()
    fmt.Println(r1.updateStatus())

    r2 := new(HttpResponse) // r2 是指针
    r2.validResponse()
    fmt.Println(r2.updateStatus())
}
/*
输出：
{200}
{200}
*/
```

validResponse()接收一个指向 HttpResponse 的指针，并改变它内部的成员；updateStatus()复制 HttpResponse 的值并只输出 HttpResponse 的内容，r1 是值而 r2 是指针。

接着，在 updateStatus()中改变接收者 r 的值，将会看到它可以正常编译，但是开始的 r 值没有被改变。因为指针作为接收者不是必需的，如下面的例子中，Point 的值仅仅用于计算：

```
type Point struct { x, y, z float }
// Point 的方法
func (p Point) Abs float {
    return math.Sqrt(p.x*p.x + p.y*p.y + p.z*p.z)
}
```

上面的写法内存占用稍微有点高，因为 Point 是作为值传递给方法的，因此传递的是它的副本，这在 Go 语言中是合法的。

为此，可以把 p 定义为一个指针来减少内存占用：

```
p := &Point{ 3, 4, 5 }
```

在后续代码中，都可以使用 p.Abs()来替代之前的(*p).Abs()这种写法。

8.3.5　方法值和方法表达式

我们在一个表达式里执行一个甚至多个方法，比如常见的 p.Distance()形式，实际上将其分成两步来执行也是可以的。其中 p.Distance 中的小数点叫作“选择器”，选择器会返回它前面那个方法（或变量）的“值”并传递给后面那个方法（Point.Distance），其中 Distance 一定是一个绑定到特定接收器变量的函数。这个函数可以不通过指定其接收器即可被调用，调用时不需要指定接收器，只要传入函数的参数即可：

```
p := Point{1, 2}
q := Point{4, 6}

distanceFromP := p.Distance        // 方法值
fmt.Println(distanceFromP(q))      // "5"
var origin Point                   // {0, 0}
fmt.Println(distanceFromP(origin)) // "2.23606797749979", sqrt(5)

scaleP := p.ScaleBy // 方法值
scaleP(2)          // (2, 4)
scaleP(3)          // (6, 12)
scaleP(10)         // (60, 120)
```

方法和函数定义语法的区别在于前者实例接收参数，编译器也是以此确定方法所属的类型。上面就是方法值的调用形式，由于方法值是一个函数，所以它肯定不支持重载，而且参数和 p.Distance 的参数一样。

但是当方法值被赋值给变量或者是作为参数传递时（例如上面的 distanceFromP 和 scaleP），会立即计算并复制该方法执行所需要的接收器对象，与其绑定，以便在稍后执行时，能隐式传递接收器对象。如果目标方法的接收器是指针类型，那么被复制的仅仅是指针，否则就是复制整个类型值。

因此，当方法或者函数类型相同时，如果需要根据其他外部环境，决定使用某个方法或者函数时，那么表达式就非常具有实用性。通过类型引用的方法表达式会被还原成为普通函数样式，接收器是第一个参数，调用时必须显式传参。至于类型，只要目标方法存在于该类型方法集中即可。如下面例子：

```
p := Point{1, 2}
q := Point{4, 6}

distance := Point.Distance   // 方法表达式
fmt.Println(distance(p, q))  // "5"
fmt.Printf("%T\n", distance) // "func(Point, Point) float64"

scale := (*Point).ScaleBy
scale(&p, 2)
fmt.Println(p)            // "{2 4}"
fmt.Printf("%T\n", scale) // "func(*Point, float64)"

// 这个 Distance 实际上是指定了 Point 对象为接收器的一个方法 func (p Point) Distance(),
// 但通过 Point.Distance 得到的函数需要比实际的 Distance 方法多一个参数，
// 即其需要用第一个额外参数指定接收器，后面排列 Distance 方法的参数。
```

在上面代码中，方法的表达式看着比方法值多了一个参数，这是因为方法的表达式会把第一个参数作为接收器，后面的参数才是函数的真正参数。

总结一下，现假设 x 的静态类型为 T，M 是 T 的方法集里面的一个方法，则 x.M 称为方法值（上面的第一种情况）。方法值是一个函数，参数和 x.M 的参数一样。T 可以是接口类型或者非接口类型。

又因为 Go 语言中，一个指针可以调用“值接收器”的非接口方法，所以 pt.M1 等价于(*pt).M1。

而一个值可以调用“指针接收器”的非接口方法，所以 s.M2 等价于(&s).M2，它会把

这个值的地址作为参数。

因此，对于非接口方法，不管它的接收器是指针类型还是值类型，值对象和指针对象都可以调用。

通过指针调用“值接收器”的方法不会改变指针指向的对象的值，因为它会复制一份值，而不是把自己的值传入方法中：

```
// 示例代码 8-15
import "fmt"
type S struct {
      Name string
}
func (s S) M1() {
      s.Name = "value"
}
func (s *S) M2() {
      s.Name = "pointer"
}
func main() {
      var s1 = S{"new"}
      var s2 = &s1
      s1.M2()
      fmt.Printf("%+v, %+v\n", s1, s2) //{Name:pointer}, &{Name:pointer}
      s1 = S{"new"}
      s2 = &s1
      s2.M1()
      fmt.Printf("%+v, %+v\n", s1, s2)//{Name:new}, &{Name:new}
}
```

8.3.6　方法和未导出字段

当类型被明确导出时，它的字段并没有被导出，例如 p.firstName 的写法就是错误的。想在另一个程序中修改或者只是读取一个类型的未导出字段可以通过面向对象语言一个众所周知的技术来完成：提供 getter 和 setter 方法。

对于 setter 方法使用 Set 前缀，对于 getter 方法使用成员名。下面是 person.go 的代码：

```
// 示例代码 8-16
package person

type Person struct {
    firstName string
    lastName  string
}
```

```
func (p *Person) FirstName() string {
    return p.firstName
}

func (p *Person) SetFirstName(newName string) {
    p.firstName = newName
}
```

上面代码设置了 FirstName 的 getter 和 setter 方法，现在如果需要改变 Person 类型中的字段的值，需要在外部调用 Setter 方法，下面是 usePerson.go 的代码，展示如何改变未导出的字段的值：

```
// 示例代码 8-17
package main

import (
    "./person"
    "fmt"
)

func main() {
    p := new(person.Person)
    // p.firstName 未定义返回错误：
    // cannot refer to unexported field or method firstName.
    // p.firstName = "Tom"
    p.SetFirstName("Tom")
    fmt.Println(p.FirstName()) // 返回：Tom
}
```

应该避免出现多个线程同时操作对象的字段的值，这会导致线程之间产生竞争并极有可能返回无法预料的结果（无法得知哪个线程最后操作了该字段的值）。如果程序执行时发生这种情况，为了安全并发访问，可以使用 Go 语言官方提供的 sync 标准库中的方法。在后面章节中会通过 goroutines 和 channels 探索并发访问对象的方式。

8.3.7 嵌入类型的方法和继承

1. 嵌入类型的方法

由于 Go 语言并不是一门传统意义上的面向对象编程语言（Java、PHP 等），所以 Go 语言无法在语言层面上直接实现类的继承和亚类的实现，但由于 Go 语言提供了创建匿名结构体的方法，所以可以把匿名结构体嵌入具名结构体内部，这样具名结构体也会拥有其内部匿名结构体的那些方法，在效果上等同于面向对象编程中的类的继承。这与 Python、

Ruby 等语言中的混入（mixin）相似。

下面是一个完整例子，它展示了内嵌结构体上的方法可以直接在外层类型的实例上调用：

```
// 示例代码 8-18
package main

import (
    "fmt"
    "math"
)
// 一个已知的具名结构体
type Point struct {
    x, y float64
}

func (p *Point) Abs() float64 {
    return math.Sqrt(p.x*p.x + p.y*p.y)
}
// NamePoint 结构体内部包含匿名字段 Point
type NamedPoint struct {
    Point
    name string
}

func main() {
    n := &NamedPoint{Point{3, 4}, "Pythongoooo"}
    fmt.Println(n.Abs()) // 打印 5
}
```

将一个已存在类型的字段和方法注入另一个类型里即为内嵌，匿名字段上的方法“晋升”成为了外层类型的方法。当然类型可以有只作用于本身实例而不作用于内嵌“父”类型上的方法，可以覆写方法（像字段一样）。

和内嵌类型方法具有同样名字的外层类型的方法会覆写内嵌类型对应的方法。例如修改代码为下面这样：

```
func (n *NamedPoint) Abs() float64 {
    return n.Point.Abs() * 100
}
```

现在 fmt.Println(n.Abs())会打印 500。

因为一个结构体可以嵌入多个匿名类型，所以实际上我们可以有一个简单版本的多重继承，就像：type Child struct { Father; Mother}。在多重继承中会进一步讨论这个问题。

结构体内嵌和自己在同一个包中的结构体时，可以彼此访问对方所有的字段和方法。

2. 多重继承

多重继承在生活中经常遇见，例如孩子继承父母的特征，父母是两个父级类。在大部分面向对象语言中，是不允许多重继承的，因为这会导致编译器变得复杂，不过由于 Go 语言并没有类的概念，所谓继承其实是内嵌结构体，通过在类型中嵌入所有必要的父类型，可以很简单地实现多重继承。Go 语言的多重继承不支持多重嵌套（即父级类型内部不允许有匿名结构体字段）。

作为一个例子，假设有一个类型 CameraPhone，通过它可以调用 Call()函数，也可以调用 TakeAPicture()函数，但是第一个方法属于类型 Phone，第二个方法属于类型 Camera。

只要嵌入这两个类型就可以解决这个问题，如下所示：

```
// 示例代码 8-19
package main

import (
    "fmt"
)

type Camera struct{}

func (c *Camera) TakeAPicture() string {
    return "拍照"
}

type Phone struct{}

func (p *Phone) Call() string {
    return "响铃"
}

type CameraPhone struct {
    Camera
    Phone
}

func main() {
    cp := new(CameraPhone)
    fmt.Println("我们的新款拍照手机有多种功能：")
    fmt.Println("打开了相机：", cp.TakeAPicture())
    fmt.Println("电话来电：", cp.Call())
}
```

```
/*
返回：
我们的新款拍照手机有多种功能：
打开了相机：拍照
电话来电：响铃
*/
```

8.4　本章小结

在本章的学习中，我们知道使用关键字 struct 或者通过指定已经存在的类型，可以声明用户定义的类型，而方法提供了一种给用户定义的类型增加行为的方式。接下来的章节中，将介绍接口与反射的操作和实现。

第9章

接口与反射

接口在 Go 语言中有着至关重要的地位，如果说 goroutine 和 channel 是支撑起 Go 语言的并发模型的基石，让 Go 语言在如今集群化与多核化的时代成为一道极为亮丽的风景，那么接口是 Go 语言整个类型系统的基石，让 Go 语言在基础编程哲学的探索上达到前所未有的高度。

9.1 接口

在编程哲学上，Go 语言站在了改革的一方，而不是改良的一方。这么说并不是因为 Go 语言有 goroutine 和 channel，而是因为 Go 语言的类型系统及 Go 语言的接口。Go 语言接口与传统编程语言的接口概念完全不一样，Go 语言的编程哲学因为有接口而趋近完美。

Go 语言的接口不只是接口，本章很大一部分都会介绍 Go 语言的接口特性，Go 语言的接口并不是其他语言（C++、Java、C#等）中所提供的接口概念。

在 Go 语言出现之前，接口主要作为不同组件之间的契约存在。对契约的实现是强制的，开发者必须声明实现该接口。为了实现一个接口，你需要从该接口继承。即使另外有一个接口 IFoo2 实现了与 IFoo 完全一样的接口方法，甚至名字也叫 IFoo，只不过两个接口分别位于不同的名字空间下，编译器也会认为上面的类 Foo 只实现了 IFoo 而没有实现 IFoo2 接口。

这类接口称为侵入式接口。“侵入式”的主要表现在于实现类需要明确声明自己实现了某个接口，这种强制性的接口继承是面向对象编程思想发展过程中一个遭受相当多置疑的特性。接下来讨论为什么这是个问题，以及为何 Go 语言的接口设计是一个更合适的选择。

9.1.1　接口是什么

Go 语言是“非传统”的面向对象编程语言，它没有类和继承的概念，但是 Go 语言里有非常灵活的接口概念，通过它可以实现很多面向对象的特性。接口提供了一种方式来说明对象的行为：如果谁能搞定这件事，它就可以在这里调用。

接口定义了一组方法（方法集），但是这些方法不包含（实现）代码——它们没有被实现（它们是抽象的），接口里也不能包含变量。

通过如下格式定义接口：

```
type Namer interface {
    Method1(param_list) return_type
    Method2(param_list) return_type
    ...
}
```

上面的 Namer 是一个接口类型。按照约定，只包含一个方法的接口的名字由方法名加[e]r 后缀组成，例如 Printer、Reader、Writer、Logger、Converter 等。还有一些不常用的方式（当后缀 er 不合适时），比如 Recoverable，此时接口名以 able 结尾，或者以 I 开头。

Go 语言中的接口都很简短，通常它们会包含 0～3 个方法。不像大多数面向对象编程语言，在 Go 语言中接口可以有值，一个接口类型的变量或一个接口值，例如 var in Namer，其中 in 是一个多字（multiword）数据结构，它的值为 nil，它本质上是一个指针，虽然不完全是一回事。指向接口值的指针是非法的，它们不仅一点用也没有，还会导致代码错误。此处的方法指针表是通过运行时反射能力构建的。

类型（比如结构体）实现接口方法集中的方法，每一个方法的实现说明了此方法是如何作用于该类型的：即实现接口，同时方法集也构成了该类型的接口。实现了 Namer 接口类型的变量可以赋值给 in（接收者值），此时方法表中的指针会指向被实现的接口方法。当然，如果另一个类型（也实现了该接口）的变量被赋值给 ai，对应指针和方法实现也会随之改变。

类型不需要显式声明它实现了某个接口：接口被隐式实现。多个类型可以实现同一个接口，实现某个接口的类型（除了实现接口方法外）可以有其他的方法。一个类型可以实现多个接口，接口类型可以包含一个实例的引用，该实例的类型实现了此接口（接口是动态类型）。

即使接口在类型实现之后才定义，二者处于不同的包中，被单独编译，只要类型实现

了接口中的方法，它就实现了此接口，所有这些特性使得接口具有很大的灵活性。

```
// 示例代码 9-1
package main

import "fmt"

type Shaper interface {
    Area() float32
}

type Square struct {
    side float32
}

func (sq *Square) Area() float32 {
    return sq.side * sq.side
}

func main() {
    sq1 := new(Square)
    sq1.side = 5

    // 定义 areaIntf
    // areaIntf = sq1
    // 更简洁，不需要分开定义：
    // areaIntf := Shaper(sq1)
    // 甚至这样：
    areaIntf := sq1
    fmt.Printf("面积为：%f\n", areaIntf.Area())
}
/*
返回：
面积为：25.000000
*/
```

上面的程序定义了一个结构体 Square 和一个接口 Shaper，接口有一个方法 Area()。在 main()方法中创建了一个 Square 的实例。在主程序外边定义了一个接收者类型是 Square 方法的 Area()，用来计算正方形的面积，结构体 Square 实现了接口 Shaper。

所以可以将一个 Square 类型的变量赋值给一个接口类型的变量：areaIntf = sq1 。

现在接口变量包含一个指向 Square 变量的引用，通过它可以调用 Square 上的方法 Area()。当然也可以直接在 Square 的实例上调用此方法，但是在接口实例上调用此方法更具通用性。接口变量里包含了接收者实例的值和指向对应方法表的指针。

这是 Go 语言版本的“多态”，多态是面向对象编程中一个广为人知的概念：根据当前的类型选择正确的方法，或者说同一种类型在不同的实例上似乎表现出不同的行为。

如果 Square 没有实现 Area()方法，编译器将会给出清晰的错误信息：

```
cannot use sq1 (type *Square) as type Shaper in assignment:
*Square does not implement Shaper (missing Area method)
```

如果 Shaper 有另外一个方法 Perimeter()，但是 Square 没有实现它，即使没有人在 Square 实例上调用这个方法，编译器也会给出上面的错误。

扩展上面的例子，类型 Rectangle 也实现了 Shaper 接口。接着创建一个 Shaper 类型的数组，迭代它的每一个元素并在上面调用 Area()方法，以此来展示多态行为：

```
// 示例代码 9-2
package main

import "fmt"

type Shaper interface {
    Area() float32
}

type Square struct {
    side float32
}

func (sq *Square) Area() float32 {
    return sq.side * sq.side
}

type Rectangle struct {
    length, width float32
}

func (r Rectangle) Area() float32 {
    return r.length * r.width
}

func main() {

    r := Rectangle{5, 3} // 需要的是一个值
    q := &Square{5}      // 传入的是指针
    // shapes := []Shaper{Shaper(r), Shaper(q)}
    // 更简洁的写法:
    shapes := []Shaper{r, q}
      // 遍历 shapes
```

```
    for n, _ := range shapes {
        fmt.Println("形状参数：", shapes[n])
        fmt.Println("形状面积是：", shapes[n].Area())
    }
}
/*
返回：
形状参数：{5 3}
形状面积是：15
形状参数：&{5}
形状面积是：25
*/
```

在调用 shapes[n].Area()时，只知道 shapes[n]是一个 Shaper 对象，最后它摇身一变成为了一个 Square 或 Rectangle 对象，并且表现出了相对应的行为。

也许从现在开始你将看到通过接口如何产生更干净、更简单及更具扩展性的代码。在后面章节将看到在开发中为类型添加新的接口是多么容易。

下面是一个更具体的例子：有两个类型 stockPosition 和 car，它们都有一个 getValue()方法，可以定义一个具有此方法的接口 valuable。接着定义一个使用 valuable 类型作为参数的函数 showValue()，所有实现了 valuable 接口的类型都可以用这个函数。

```
// 示例代码 9-3
package main

import "fmt"

type stockPosition struct {
    ticker     string
    sharePrice float32
    count      float32
}

/* 获取 stock 的值（价格） */
func (s stockPosition) getValue() float32 {
    return s.sharePrice * s.count
}

type car struct {
    make  string
    model string
    price float32
}

/* 获取 car 的值（价格） */
func (c car) getValue() float32 {
```

```
    return c.price
}

/* 定义具有价值的不同事物的“合同” */
type valuable interface {
    getValue() float32
}

func showValue(asset valuable) {
    fmt.Printf("资产的价值是: %f\n", asset.getValue())
}

func main() {
    var o valuable = stockPosition{"GOOG", 577.20, 4}
    showValue(o)
    o = car{"BMW", "M3", 66500}
    showValue(o)
}
/*
返回:
资产的价值是: 2308.800049
资产的价值是: 66500.000000
*/
```

举一个标准库的例子，io 包里有一个接口类型 Reader:

```
type Reader interface {
    Read(p []byte) (n int, err error)
}
```

定义变量 var r io.Reader，那么就可以写如下的代码：

```
    var r io.Reader
    r = os.Stdin
    r = bufio.NewReader(r)
    r = new(bytes.Buffer)
    f,_ := os.Open("test.txt")
    r = bufio.NewReader(f)
```

上面 r 右边的类型都实现了 Read()方法，并且有相同的方法签名，r 的静态类型是 io.Reader。

注意，有的时候，也会以一种稍微不同的方式来使用接口这个词，从某个类型的角度来看，它的接口指的是：它的所有导出方法，只不过没有显式地为这些导出方法额外指定一个接口而已。

9.1.2 接口类型与约定

接口类型实际上是描述了一系列方法的集合，一个实现了这些方法的具体类型是这个接口类型的实例。

io.Writer 类型是用得最广泛的接口之一，因为它提供了所有的类型写入 bytes（字节）的抽象，包括文件类型、内存缓冲区、网络链接、HTTP 客户端、压缩工具、哈希等。io 包中定义了很多其他有用的接口类型。io.Reader 可以代表任意可读取 bytes 的类型，io.Closer 可以是任意可关闭的值，例如一个文件或网络链接：

```
package io

type Reader interface {
      Read(p []byte) (n int, err error)
}
type Closer interface {
      Close() error
}
```

再往下看，我们发现有些新的接口类型通过组合已经有了接口来定义，下面是两个例子：

```
type ReadWriter interface {
      Reader
      Writer
}
type ReadWriteCloser interface {
      Reader
      Writer
      Closer
}
```

上面用到的语法和结构内嵌相似，可以用这种方式以一个简写命名另一个接口，而不用声明它所有的方法，这种方式被称为接口内嵌。尽管略失简洁性，可以像下面这样，不使用内嵌来声明 io.Writer 接口：

```
type ReadWriter interface {
      Read(p []byte) (n int, err error)
      Write(p []byte) (n int, err error)
}
```

或者可以使用混合的风格：

```
type ReadWriter interface {
      Read(p []byte) (n int, err error)
```

```
    Writer
}
```

上面 3 种定义方式都是一样的效果。方法的顺序变化也没有影响，唯一重要的就是这个集合里面的方法。

1. 动态类型

一个接口类型的变量 varI 中可以包含任何类型的值，必须有一种方式来检测它的动态类型，即运行时在变量中存储的值的实际类型。在执行过程中动态类型可能会有所不同，但是它一定是可以分配给接口变量的类型。通常可以使用类型断言（Go 语言内置的一种智能推断类型的功能）来测试在某个时刻 varI 是否包含类型 T 的值：

```
v := varI.(T)       // 未经检查的类型断言
```

varI 必须是一个接口变量，否则编译器会报错：invalid type assertion: varI.(T) (non-interface type (type of varI) on left) 。

类型断言可能是无效的，虽然编译器会尽力检查转换是否有效，但是它不可能预见所有的可能性。如果转换在程序运行时失败，则会导致错误发生。更安全的方式是使用以下形式来进行类型断言：

```
if v, ok := varI.(T); ok {  // 已检查类型断言
    Process(v)
    return
}
// varI 不是类型 T
```

如果转换合法，v 是 varI 转换到类型 T 的值，ok 会是 true；否则 v 是类型 T 的零值，ok 是 false，也没有运行时错误发生。应该使用这种方式来进行类型断言。

在多数情况下，可能只是想在 if 中测试 ok 的值，此时使用以下的方法会是最方便的：

```
if _, ok := varI.(T); ok {
    // ...
}
```

具体示例如下：

```
// 示例代码 9-4
package main

import (
    "fmt"
    "math"
)
```

```
type Square struct {
    side float32
}

type Circle struct {
    radius float32
}

type Shaper interface {
    Area() float32
}

func main() {
    var areaIntf Shaper
    sq1 := new(Square)
    sq1.side = 5

    areaIntf = sq1
    // areaIntf 的类型是否是 Square
    if t, ok := areaIntf.(*Square); ok {
        fmt.Printf("areaIntf 的类型是: %T\n", t)
    }
    if u, ok := areaIntf.(*Circle); ok {
        fmt.Printf("areaIntf 的类型是: %T\n", u)
    } else {
        fmt.Println("areaIntf 不含类型为 Circle 的变量")
    }
}

func (sq *Square) Area() float32 {
    return sq.side * sq.side
}

func (ci *Circle) Area() float32 {
    return ci.radius * ci.radius * math.Pi
}
/*
返回:
areaIntf 的类型是: *main.Square
areaIntf 不含类型为 Circle 的变量
*/
```

程序行中定义了一个新类型 Circle，它也实现了 Shaper 接口。

第一个 if 语句测试 areaIntf 里是否包含一个 Square 类型的变量，返回的 ok 为 true 则表示包含该类型；然后第二个 if 语句测试它是否包含一个 Circle 类型的变量，返回的 OK

结果为 false，所以不包含该类型。

如果忽略 areaIntf.(*Square)中的*号，会导致编译错误：

```
impossible type assertion: Square does not implement Shaper (Area method has pointer
receiver)
```

这是因为 Go 语言编译器无法自动推断类型，Area()方法通过指针接收器传入参数。

2．类型判断

接口变量的类型也可以使用 type-swtich 来检测（部分代码）：

```
switch t := areaIntf.(type) {
case *Square:
    fmt.Printf("Square 类型的 %T 值为：%v\n", t, t)
case *Circle:
    fmt.Printf("Circle 类型的 %T 值为：%v\n", t, t)
case nil:
    fmt.Printf("nil 值：发生了意外。\n")
default:
    fmt.Printf("未知类型 %T\n", t)
}
/*
返回：
Square 类型的 *main.Square 值为：&{5}
*/
```

变量 t 得到了 areaIntf 的值和类型，所有 case 语句中列举的类型（nil 除外）都必须实现对应的接口（在上例中即 Shaper），如果被检测类型没有在 case 语句列举的类型中，就会执行 default 语句。

可以用 type-switch 进行运行时类型分析，但是 type-switch 不允许有 fallthrough。如果仅仅是测试变量的类型，不用它的值，那么就可以不需要赋值语句，比如：

```
switch areaIntf.(type) {
case *Square:
    // ...
case *Circle:
    // ...
...
default:
    // ...
}
```

下面的代码片段展示了一个类型分类函数，它有一个可变长度参数，可以是任意类型的数组，它会根据数组元素的实际类型执行不同的动作：

```
func classifier(items ...interface{}) {
    for i, x := range items {
        switch x.(type) {
        case bool:
            fmt.Printf("参数 #%d 类型是 bool\n", i)
        case float64:
            fmt.Printf("参数 #%d 类型是 float64\n", i)
        case int, int64:
            fmt.Printf("参数 #%d 类型是 int\n", i)
        case nil:
            fmt.Printf("参数 #%d 类型是 nil\n", i)
        case string:
            fmt.Printf("参数 #%d 类型是 string\n", i)
        default:
            fmt.Printf("参数 #%d 类型未知\n", i)
        }
    }
}
```

可以这样调用此方法：classifier(13, -14.3, "BELGIUM", complex(1, 2), nil, false)。在处理来自外部的、类型未知的数据时，比如解析诸如 JSON 或 XML 编码的数据，类型测试和转换会非常有用。

9.1.3 接口实现

一个类型如果拥有一个接口需要的所有方法，那么这个类型就实现了这个接口。例如，*os.File 类型实现了 io.Reader、Writer、Closer 和 ReadWriter 接口。

*bytes.Buffer 实现了 Reader、Writer 和 ReadWriter 这些接口，但是它没有实现 Closer 接口，所以它不具有 Close 方法。Go 语言的程序员经常会简要地把一个具体的类型描述成一个特定的接口类型。

举个例子，*bytes.Buffer 是 io.Writer；*os.Files 是 io.ReadWriter。接口指定的规则非常简单：表达一个类型属于某个接口只要这个类型实现这个接口就可以，这个规则甚至适用于等式右边本身也是一个接口类型的情况，因为 ReadWriter 和 ReadWriteCloser 包含所有 Writer 的方法，所以任何实现了 ReadWriter 和 ReadWriteCloser 的类型必定也实现了 Writer 接口。

在进一步学习前，必须先解释表示一个类型持有一个方法当中的细节。对于每一个命名过的具体类型 T，它的一些方法的接收者是类型 T 本身，然而另一些则是一个 T 的指针。还记得在 T 类型的参数上调用一个 T 的方法是合法的，只要这个参数是一个变量。

编译器隐式地获取了它的地址，但这仅仅是一个语法糖：T 类型的值不拥有所有*T 指针的方法，那这样它就可能只实现更少的接口。

举个例子可能会更清晰一点，IntSet 类型的 String 方法的接收者是一个指针类型，所以我们不能在一个不能寻址的 IntSet 值上调用这个方法：

```
type IntSet struct {
    /* .... */
}
func (*IntSet) String() string
var _ = IntSet{}.String()
// 编译错误返回：String requires *IntSet receiver
```

但是可以在一个 IntSet 值上调用这个方法：

```
var s IntSet
var _ = s.String()
// 编译通过：s 是一个变量，&s 有一个 String 方法
```

然而，由于只有 IntSet 类型有 String 方法，所有也只有 IntSet 类型实现了 fmt.Stringer 接口：

```
var _ fmt.Stringer = &s // 编译通过
var _ fmt.Stringer = s  // 编译错误返回：IntSet lacks String method
```

下面是接口在 runtime 中的实现，注意其中包含了接口本身和实际数据类型的类型信息：

```
// src/runtime/runtime2.go
type iface struct {
    // 包含接口的静态类型信息、数据的动态类型信息和函数表
    tab *itab
    /* 指向具体数据的内存地址，比如 slice、map 等，或者在接口转换时直接存放小数据(一个
    指针的长度) */
    data unsafe.Pointer
}
type itab struct {
    // 接口的类型信息
    inter *interfacetype
    // 具体数据的类型信息
    _type  *_type
    link   *itab
    hash   uint32
    bad    bool
    inhash bool
    unused [2]byte
    /* 函数地址表，这里放置和接口方法对应的具体数据类型的方法地址
       实现接口调用方法的动态分派，一般在每次给接口赋值发生转换时
```

```
        会更新此表，或者直接获取缓存的 itab */
    fun [1]uintptr // 变量大小
}
```

另外，需要注意与接口相关的两点优化，会影响到反射等的实现。

（1）空接口（interface{}）的 itab 优化：当将某个类型赋值给空接口时，由于空接口没有方法，所以上面空接口 iface 的 tab 字段会直接指向数据的具体类型。在 Go 语言的 reflect 包中，reflect.TypeOf 和 reflect.ValueOf 的参数都是空接口，因此所有参数都会先转换为空接口类型，这样反射就实现了对所有参数类型获取实际数据类型的统一，这在后面反射的基本实现中会分析到。

（2）发生“接口转换”时 data 字段相关的优化：当被转换为接口的数据的类型长度不超过一个指针的长度时（比如 pointer、map、func、chan、[1]int 等类型），接口转换时会将数据直接复制到接口的 data 字段（DirectIface）中，而不再额外分配内存并复制。另外，从 Go 1.8+的源码来看除 DirectIface 的优化以外，还对长度较小（不超过 64B，未初始化数据内存的 array、空字符串等）的零值做了优化，也不会重新分配内存，而是直接指向一个包级全局数组变量 zeroVal 的首地址。注意这里的优化发生在接口转换时生成的临时接口上，而不是被赋值的接口左值上。

再者，在 Go 语言中只有值传递（包括接口类型），与具体的类型实现无关，但是某些类型具有引用的属性。典型的 9 种非基础类型中：

- array 传递会复制整块数据内存，传递长度为 len(arr) * Sizeof(elem)。
- string、slice、interface 传递的是其 runtime 的实现，所以长度是固定的，分别为 16B、24B、16B（AMD64）。
- map、func、chan、pointer 传递的是指针，所以长度固定为 8B（AMD64）。
- struct 传递的是所有字段的内存副本，所以长度是所有字段的长度之和。

9.1.4 嵌套接口

一个接口可以包含一个或多个其他接口，这相当于直接将这些内嵌接口的方法列举在外层接口中。比如接口 File 包含了 ReadWrite 和 Lock 的所有方法，它还额外有一个 Close() 方法：

```
type ReadWrite interface {
    Read(b Buffer) bool
    Write(b Buffer) bool
```

```
}

type Lock interface {
    Lock()
    Unlock()
}

type File interface {
    ReadWrite
    Lock
    Close()
}
```

9.1.5　接口赋值

接口赋值在 Go 语言中分为如下两种情况：将对象实例赋值给接口；将一个接口赋值给另一个接口。

先讨论将某种类型的对象实例赋值给接口，这要求该对象实例实现了接口要求的所有方法，如下：

```
type Integer int

func (a Integer) Less(b Integer) bool {
      return a < b
}
func (a *Integer) Add(b Integer) {
      *a += b
}
```

相应地，定义接口 LessAdder：

```
type LessAdder interface {
    Less(b Integer) bool
    Add(b Integer)
}
```

现在有个问题：假设定义一个 Integer 类型的对象实例，怎样将其赋值给 LessAdder 接口呢？应该用下面的语句 1，还是语句 2 呢？

```
var a Integer = 1
var b LessAdder = &a // 语句 1
var b LessAdder = a  // 语句 2
```

答案是应该用语句 1，原因在于，Go 语言可以根据函数 func (a Integer) Less(b Integer) bool 自动生成一个新的 Less()方法：

仅供非商业用途或交流学习使用




```
func (a *Integer) Less(b Integer) bool {
      return (*a).Less(b)
}
```

这样，类型*Integer 就既包含 Less()方法，也包含 Add()方法，满足 LessAdder 接口。而从另一方面来说，根据 func (a *Integer) Add(b Integer)函数无法自动生成以下这个成员方法：

```
func (a Integer) Add(b Integer) {
      (&a).Add(b)
}
```

因为(&a).Add()改变的只是函数参数 a，对外部实际要操作的对象并无影响，这不符合用户的预期。所以，Go 语言不会自动为其生成该函数。因此，类型 Integer 只存在 Less()方法，缺少 Add()方法，不满足 LessAdder 接口，故上面的语句 2 不能赋值。

为了进一步证明以上的推理，不妨再定义一个 Lesser 接口：

```
type Lesser interface {
      Less(b Integer) bool
}
```

然后定义一个 Integer 类型的对象实例，将其赋值给 Lesser 接口：

```
var a Integer = 1
var b1 Lesser = &a   // 语句 1
var b2 Lesser = a    // 语句 2
```

正如我们所料，语句 1 和语句 2 均可以编译通过。我们再来讨论另一种情形：将一个接口赋值给另一个接口。在 Go 语言中，只要两个接口拥有相同的方法列表（次序不同不要紧），那么它们就是等同的，可以相互赋值。

下面来看一个示例，这是第一个接口：

```
package one
type ReadWriter interface {
    Read(buf []byte) (n int, err error)
    Write(buf []byte) (n int, err error)
}
```

第二个接口位于另一个包中：

```
package two
type IStream interface {
    Write(buf []byte) (n int, err error)
    Read(buf []byte) (n int, err error)
}
```

仅供非商业用途或交流学习使用



这里定义了两个接口，一个叫 one.ReadWriter，另一个叫 two.IStream，两者都定义了 Read()、Write()方法，只是定义次序相反。one.ReadWriter 先定义了 Read()再定义了 Write()，而 two.IStream 反之。

在 Go 语言中，这两个接口实际上并无区别，因为任何实现了 one.ReadWriter 接口的类，均实现了 two.IStream；任何 one.ReadWriter 接口对象可赋值给 two.IStream，反之亦然；在任何地方使用 one.ReadWriter 接口与使用 two.IStream 接口并无差异。

以下这些代码可编译通过:

```
var file1 two.IStream = new(File)
var file2 one.ReadWriter = file1
var file3 two.IStream = file2
```

接口赋值并不要求两个接口必须等价。如果接口 A 的方法列表是接口 B 的方法列表的子集，那么接口 B 可以赋值给接口 A。例如，假设有 Writer 接口：

```
type Writer interface {
    Write(buf []byte) (n int, err error)
}
```

就可以将上面的 one.ReadWriter 和 two.IStream 接口的实例赋值给 Writer 接口：

```
var file1 two.IStream = new(File)
var file4 Writer = file1
```

但是反过来并不成立：

```
var file1 Writer = new(File)
var file5 two.IStream = file1 // 编译不能通过
```

这段代码无法编译通过，原因很显然：file1 并没有 Read()方法。

9.1.6　接口查询

有办法让上面的 Writer 接口转换为 two.IStream 接口吗？有，那就是即将讨论的接口查询语法，代码如下：

```
var file1 Writer = ...
if file5, ok := file1.(two.IStream); ok {
      ...
}
```

这个 if 语句检查 file1 接口指向的对象实例是否实现了 two.IStream 接口，如果实现了，则执行特定的代码。

仅供非商业用途或交流学习使用



接口查询是否成功，要在运行期才能确定。它不像接口赋值，编译器只需通过静态类型检查即可判断赋值是否可行。在 Go 语言中，对象是否满足某个接口，通过某个接口查询其他接口，这一切都是完全自动完成的。

让语言内置接口查询，这是一件非常了不起的事情。在 Go 语言中，可以询问接口指向的对象是否是某个类型，比如：

```
var file1 Writer = ...
if file6, ok := file1.(*File); ok {
      ...
}
```

这个 if 语句判断 file1 接口指向的对象实例是否是*File 类型，如果是则执行特定代码。查询接口所指向的对象是否为某个类型的这种用法可以认为只是接口查询的一个特例。

9.1.7 接口组合

像之前介绍的类型组合一样，Go 语言同样支持接口组合。我们已经介绍过 Go 语言包中 io.Reader 接口和 io.Writer 接口，接下再介绍同样来自 io 包的另一个接口 io.ReadWriter：

```
// ReadWriter 接口将基本的 Read 和 Write 方法组合起来
type ReadWriter interface {
      Reader
      Writer
}
```

这个接口组合了 Reader 和 Writer 两个接口，它完全等同于如下写法：

```
type ReadWriter interface {
      Read(p []byte) (n int, err error)
      Write(p []byte) (n int, err error)
}
```

因为这两种写法的表意完全相同：ReadWriter 接口既能做 Reader 接口的所有事情，又能做 Writer 接口的所有事情。在 Go 语言包中，还有众多类似的组合接口，比如 ReadWriteCloser、ReadWriteSeeker、ReadSeeker 和 WriteCloser 等。

可以认为接口组合是类型匿名组合的一个特定场景，只不过接口只包含方法，而不包含任何成员变量。

仅供非商业用途或交流学习使用



9.2 反射

反射是用程序检查代码中所拥有的结构尤其是类型的一种能力，这是元编程的一种形式。反射可以在运行时检查类型和变量，例如它的大小、方法和动态调用这些方法。这对于没有源代码的包尤其有用。这是一个强大的工具，但是也存在一定的隐患，除非真的有必要，否则应当避免使用或小心使用。

9.2.1 方法和类型的反射

变量的最基本信息就是类型和值：反射包的 Type 用来表示一个 Go 语言类型，反射包的 Value 为 Go 值提供了反射接口。

两个简单的函数：reflect.TypeOf 和 reflect.ValueOf，返回被检查对象的类型和值。例如，x 被定义为 var x float64 = 3.4，那么 reflect.TypeOf(x)返回 float64，reflect.ValueOf(x)返回<float64 Value> 。

实际上，反射通过检查一个接口的值，把变量转换成空接口。这从下面两个函数签名（定义了函数或方法的输入与输出的格式）能够很明显看出来：

```
func TypeOf(i interface{}) Type
func ValueOf(i interface{}) Value
```

接口的值包含一个 type 和 value。反射可以从接口值反射到对象，也可以从对象反射回接口值。

reflect.Type 和 reflect.Value 都有许多方法用于检查和操作它们。一个重要的例子是 Value 有一个 Type 方法返回 reflect.Value 的 Type；另一个是 Type 和 Value 都有 Kind 方法返回一个常量来表示类型：Uint、Float64、Slice 等。同样 Value 有叫作 Int 和 Float 的方法可以获取存储在内部的值（和 int64 和 float64 一样）。

对于变量 x，如果 v:=reflect.ValueOf(x)，那么 v.Kind()返回 float64，所以下面的表达式返回的结果是 true（即两者全等）：

```
v.Kind() == reflect.Float64
```

Kind()总是返回底层类型：

```
type MyInt int
var m MyInt = 5
v := reflect.ValueOf(m)
```

仅供非商业用途或交流学习使用



方法 v.Kind()返回 reflect.Int。变量 v 的 Interface()方法可以得到还原（接口）值，所以可以这样打印 v 的值：fmt.Println(v.Interface())。

尝试运行下面的代码：

```
// 示例代码 9-5
package main

import (
    "fmt"
    "reflect"
)

func main() {
    var x float64 = 3.4
    fmt.Println("type:", reflect.TypeOf(x))
    v := reflect.ValueOf(x)
    fmt.Println("value:", v)
    fmt.Println("type:", v.Type())
    fmt.Println("kind:", v.Kind())
    fmt.Println("value:", v.Float())
    fmt.Println(v.Interface())
    fmt.Printf("value is %5.2e\n", v.Interface())
    y := v.Interface().(float64)
    fmt.Println(y)
}
/*
返回:
type: float64
value: <float64 Value>
type: float64
kind: float64
value: 3.4
3.4
value is 3.40e+00
3.4
*/
```

x 是一个 float64 类型的值，reflect.ValueOf(x).Float()返回这个 float64 类型的实际值；reflect.ValueOf(x)适用于 Int()、Bool()、Complex()、String()等其他类型。

9.2.2 通过反射修改设置值

假设要把下面例子中 x 的值改为 3.1415，Value 有一些方法可以完成这个任务，但是必须小心使用 v.SetFloat(3.1415)，这将产生一个错误：reflect.Value.SetFloat using unaddressable value。

仅供非商业用途或交流学习使用


```
// 示例代码 9-6
package main

import (
      "fmt"
      "reflect"
)

func main() {
      var x float64 = 3.1415
      v := reflect.ValueOf(x)
      // 设置一个值：
      // v.SetFloat(3.1415)
      // 上面代码会产生错误，返回：
      // Panic: reflect.Value.SetFloat using unaddressable value
      fmt.Println("v：", v.CanSet())
      v = reflect.ValueOf(&x) // 注意：取 x 的地址
      fmt.Println("v 的类型：", v.Type())
      fmt.Println("v：", v.CanSet())
      v = v.Elem()
      fmt.Println("v 的 Elem 是：", v)
      fmt.Println("v：", v.CanSet())
      v.SetFloat(3.1415) // 编译通过
      fmt.Println(v.Interface())
      fmt.Println(v)
}
```

原因是 v 不是可设置的（这里并不是说值不可寻址），是否可设置是 Value 的一个属性，并且不是所有的反射值都有这个属性：可以使用 CanSet()方法测试是否可设置。

在例子中可以看到 v.CanSet()返回 false：settability of v: false，当 v := reflect.ValueOf(x)函数通过传递一个 x 副本创建了 v 时，那么 v 的改变并不能更改原始的 x。要想 v 的更改能作用到 x，那就必须传递 x 的地址 v = reflect.ValueOf(&x)。

通过 Type()可以看到 v 现在的类型是*float64，并且仍然是不可设置的。要想让其可设置需要使用 Elem()函数，这将间接地使用指针 v = v.Elem()，现在 v.CanSet()返回 true，并且 v.SetFloat(3.1415)设置成功了！

```
// 示例代码 9-7
package main

import (
    "fmt"
    "reflect"
)
```

仅供非商业用途或交流学习使用


```
func main() {
    var x float64 = 3.4
    v := reflect.ValueOf(x)
    // 设置一个值:
    // v.SetFloat(3.1415)
      // 同样是错误, Panic: reflect.Value.SetFloat using unaddressable value
    fmt.Println("v: ", v.CanSet())
    v = reflect.ValueOf(&x) // 注意: 取 x 的地址
    fmt.Println("v 的类型: ", v.Type())
    fmt.Println("v: ", v.CanSet())
    v = v.Elem()
    fmt.Println("v 的 Elem 是: ", v)
    fmt.Println("v: ", v.CanSet())
    v.SetFloat(3.1415) // 通过
    fmt.Println(v.Interface())
    fmt.Println(v)
}
/*
返回:
v: false
v 的类型是: *float64
v: false
v 的 Elem 是: <float64 Value>
v: true
3.1415
<float64 Value>
*/
```

反射中有些内容是需要用地址去改变它的状态的。

9.2.3 反射结构

有些时候需要反射一个结构类型：NumField()方法返回结构内的字段数量；通过一个 for 循环用索引取得每个字段的值 Field(i)。

我们同样能够调用已经定义的函数在结构上的方法，例如，使用索引 n 来调用 Method(n).Call(nil)：

```
// 示例代码 9-8
package main

import (
    "fmt"
    "reflect"
)

type NotknownType struct {
```

仅供非商业用途或交流学习使用


```
    s1, s2, s3 string
}

func (n NotknownType) String() string {
    return n.s1 + " - " + n.s2 + " - " + n.s3
}

var secret interface{} = NotknownType{"Google", "Go", "Code"}

func main() {
    value := reflect.ValueOf(secret) // <main.NotknownType Value>
    typ := reflect.TypeOf(secret)    // main.NotknownType
    // 也可以这样写:
    // typ := value.Type()  // main.NotknownType
    fmt.Println(typ)
    knd := value.Kind() // struct
    fmt.Println(knd)

    // 迭代结构的字段:
    for i := 0; i < value.NumField(); i++ {
        fmt.Printf("Field %d: %v\n", i, value.Field(i))
        // 返回错误:
      // Panic: reflect.Value.SetString using value obtained using unexported field
        // value.Field(i).SetString("C#")
    }

      // 调用第一个方法，即 string():
    results := value.Method(0).Call(nil)
    fmt.Println(results) // [Google - Go - Code]
}
/*
返回:
main.NotknownType
struct
Field 0: Google
Field 1: Go
Field 2: Code
[Google - Go - Code]
*/
```

但是如果尝试更改一个值，会得到一个错误：

```
panic: reflect.Value.SetString using value obtained using unexported field
```

这是因为结构中只有被导出字段（首字母大写）才是可设置的，来看下面的例子：

```
// 示例代码 9-9
package main
```

仅供非商业用途或交流学习使用


```
import (
    "fmt"
    "reflect"
)

type T struct {
    A int
    B string
}

func main() {
    t := T{23, "Hello"}
    s := reflect.ValueOf(&t).Elem()
    typeOfT := s.Type()
    for i := 0; i < s.NumField(); i++ {
        f := s.Field(i)
        fmt.Printf("%d: %s %s = %v\n", i,
            typeOfT.Field(i).Name, f.Type(), f.Interface())
    }
    s.Field(0).SetInt(77)
    s.Field(1).SetString("The World")
    fmt.Println("t 现在的值是：", t)
}
/*
返回：
0: A int = 23
1: B string = Hello
t 现在的值是：{77 The World}
*/
```

9.2.4 Printf 和反射

在 Go 语言的标准库中，前面所述的反射的功能被大量地使用。举个例子，fmt 包中的 Printf（以及其他格式化输出函数）都会使用反射来分析它的...参数。Printf 的函数声明为：

```
func Printf(format string, args ... interface{}) (n int, err error)
```

Printf 中的...参数为空接口类型，Printf 使用反射包来解析这个参数列表，所以，Printf 能够知道它每个参数的类型。格式化字符串中只有%d 而没有%u 和%ld，因为它知道这个参数是 unsigned 还是 long，这也是为什么 Print 和 Println 在没有格式字符串的情况下还能如此准确地输出。

为了让大家更加具体地了解 Printf 中的反射，下面实现一个简单的通用输出函数，其

中使用了 type-switch 来推导参数类型，并根据类型来输出每个参数的值：

```
// 示例代码 9-10
package main

import (
    "os"
    "strconv"
)

type Stringer interface {
    String() string
}

type Celsius float64

func (c Celsius) String() string {
    return strconv.FormatFloat(float64(c),'f', 1, 64) + " °C"
}

type Day int

var dayName = []string{"星期一", "星期二", "星期三", "星期四", "星期五", "星期六", "星
期天"}

func (day Day) String() string {
    return dayName[day]
}

func print(args ...interface{}) {
    for i, arg := range args {
        if i > 0 {os.Stdout.WriteString(" ")}
        switch a := arg.(type) { // 类型转换
            case Stringer:  os.Stdout.WriteString(a.String())
            case int:       os.Stdout.WriteString(strconv.Itoa(a))
            case string:    os.Stdout.WriteString(a)
            // 更多类型
            default:        os.Stdout.WriteString("???")
        }
    }
}

func main() {
    print(Day(1), "温度是", Celsius(18.36))  // 星期二 温度是 18.4 °C
}
```

在标准库的章节还会阐释 fmt.Fprintf()是怎样运用这种反射机制的。

9.3 本章小结

本章需要注意设计类型时确认类型的本质是原始的还是非原始的。接口是声明了一组行为并支持多态的类型，嵌入类型提供了扩展类型的能力，而无须使用继承。

结束本章的学习，意味着我们已经把 Go 语言所有的基础内容都认识了一遍，其中本章的接口和反射实际上灵活运用了很多以前的知识，换句话来说，Go 语言很多上层的数据结构与类型都是依靠接口与反射来实现的，所以本章的内容不容忽视。从下一章开始，将接触 Go 语言的另一大特性——并发编程。

第 10 章
并发编程

因为 Go 语言从语言层面支持了并行，所以有人把 Go 语言比作 21 世纪的 C 语言。通常程序会被编写为一个顺序执行并完成一个独立任务的代码，因为这种类型的程序很容易写，也很容易维护。不过也有一些情况下，并行执行多个任务会有更大的好处。

例如，Web 服务需要在各自独立的 socket 上同时接收多个数据请求。每个 socket 请求都是独立的，可以完全独立于其他 socket 进行处理，具有并行执行多个请求的能力可以显著提高这类系统的性能。需要考虑到这一点，Go 语言的语法和运行时直接内置了对并发的支持。

Go 语言里的并发指的是能让某个函数独立于其他函数运行的能力。当一个函数创建为协程（goroutine）时，Go 语言会将其视为一个独立的工作单元，这个单元会被调度到可用的逻辑处理器上执行。

对于没有使用过通道编写并发程序的程序员来说，通道会让他们感觉神奇而兴奋，希望读者使用后也能有这种感觉。使用通道可以使编写并发程序更容易，也能够让并发程序出错更少。

10.1 并发编程基础

10.1.1 并发与并行

让我们先来学习抽象程度较高的概念：什么是操作系统的线程（thread）和进程（process）。这会有助于后面理解 Go 语言运行时调度器如何利用操作系统来并发运行

goroutine。当运行一个应用程序（如一个 IDE 或者编辑器）的时候，操作系统会为这个应用程序启动一个进程，可以将这个进程看作包含了应用程序在运行中需要用到和维护的各种资源的容器。这些资源包括但不限于内存地址空间、文件和设备的句柄以及线程。一个线程是一个执行空间，这个空间会被操作系统调度来运行函数中所写的代码。每个进程至少包含一个线程，每个进程的初始线程被称作主线程。因为执行这个线程的空间是应用程序本身的空间，所以当主线程终止时，应用程序也会终止。操作系统将线程调度到某个处理器上运行，这个处理器并不一定是进程所在的处理器。不同操作系统使用的线程调度算法一般都不一样，但是这种不同会被操作系统屏蔽，并不会展示给程序员。

操作系统会在物理处理器上调度线程来运行，而 Go 语言的运行时会在逻辑处理器上调度 goroutine 来运行。每个逻辑处理器都分别绑定到单个操作系统线程。Go 语言的运行时默认会为每个可用的物理处理器分配一个逻辑处理器，这些逻辑处理器会用于执行所有被创建的 goroutine。即便只有一个逻辑处理器，Go 语言也可以以神奇的效率和性能，并发调度无数个 goroutine。

如果创建一个 goroutine 并准备运行，这个 goroutine 就会被放到调度器的全局运行队列中。之后，调度器就将这些队列中的 goroutine 分配给一个逻辑处理器，并放到这个逻辑处理器对应的本地运行队列中。本地运行队列中的 goroutine 会一直等待直到自己被分配的逻辑处理器执行。

有时，正在运行的 goroutine 需要执行一个阻塞的系统调用，如打开一个文件。当这类调用发生时，线程和 goroutine 会从逻辑处理器上分离，该线程会继续阻塞，等待系统调用的返回。与此同时，这个逻辑处理器就失去了用来运行的线程，所以，调度器会创建一个新线程，并将其绑定到该逻辑处理器上。之后，调度器会从本地运行队列里选择另一个 goroutine 来运行。一旦被阻塞的系统调用执行完成并返回，对应的 goroutine 会放回到本地运行队列，而之前的线程会保存好，以便之后可以继续使用。

如果一个 goroutine 需要做一个网络 I/O 调用，流程上会有些不一样。在这种情况下，goroutine 会和逻辑处理器分离，并移到集成了网络轮询器的运行时。一旦该轮询器指示某个网络读或者写操作已经就绪，对应的 goroutine 就会重新分配到逻辑处理器上来完成操作。调度器对可以创建的逻辑处理器的数量没有限制，但语言运行时默认限制每个程序最多创建 10 000 个线程，这个限制值可以通过调用 runtime/debug 包的 SetMaxThreads 方法来更改。如果程序试图使用更多的线程，就会崩溃。

并发（concurrency）不是并行（parallelism）。并行是让不同的代码片段同时在不同的物理处理器上执行。并行的关键是同时做很多事情，而并发是指同时管理很多事情，这些事情可能只做了一半就被暂停去做别的事情了。在很多情况下，并发的效果比并行好，因为操作系统和硬件的总资源一般很少，但能支持系统同时做很多事情。这种“使用较少的资源做更多的事情”的哲学，也是指导 Go 语言设计的哲学。

如果希望让 goroutine 并行，必须使用一个以上逻辑处理器。当有多个逻辑处理器时，调度器会将 goroutine 平等分配到每个逻辑处理器上，这会让 goroutine 在不同的线程上运行。不过要想真的实现并行的效果，需要让自己的程序运行在有多个物理处理器的机器上。否则，哪怕 Go 语言运行时使用多个线程，goroutine 依然会在同一个物理处理器上并发运行，达不到并行的效果。

调度器包含一些聪明的算法，这些算法会随着 Go 语言的发布被更新和改进，所以不推荐盲目修改语言运行时对逻辑处理器的默认设置。如果真的认为修改逻辑处理器的数量可以改进性能，也可以对语言运行时的参数进行细微调整。后面会介绍如何做这种修改。

10.1.2　指定使用核心数

Go 语言默认会调用 CPU 核心数，这些功能都是自动调整的，但是也提供了相应的标准库来指定核心数。使用 flags 包可以调整程序运行时调用的 CPU 核心数，如下：

```
var numCores = flag.Int("n", 2, "CPU 核心数")

in main()
flag.Pars()
runtime.GOMAXPROCS(*numCores)
```

协程可以通过调用 runtime.Goexit()来停止，尽管这样做几乎没有必要：

```
// 示例代码 10-1
package main

import (
    "fmt"
    "time"
)

func main() {
    fmt.Println("这里是 main()开始的地方")
    go longWait()
    go shortWait()
```

```
    fmt.Println("挂起 main()")
    // 挂起工作时间以纳秒（ns）为单位
    time.Sleep(10 * 1e9)
    fmt.Println("这里是 main()结束的地方")
}

func longWait() {
    fmt.Println("开始 longWait()")
    time.Sleep(5 * 1e9) // 等待 5 秒
    fmt.Println("结束 longWait()")
}

func shortWait() {
    fmt.Println("开始 shortWait()")
    time.Sleep(2 * 1e9) // 等待 2 秒
    fmt.Println("结束 shortWait()")
}
/*
返回:
这里是 main()开始的地方
挂起 main()
开始 longWait()
开始 shortWait()
结束 shortWait()
结束 longWait()
这里是 main()结束的地方 // 过了 10 秒
*/
```

main()、longWait()和 shortWait()三个函数作为独立的处理单元按顺序启动，然后开始并行运行，每一个函数都在运行的开始和结束阶段输出了消息。为了模拟它们运算的时间消耗，使用了 time 包中的 Sleep 函数。Sleep()可以按照指定的时间来暂停函数或协程的执行，这里使用了纳秒（ns，符号 1e9 表示 1 乘以 10 的 9 次方，e=指数）。

它们按照我们期望的顺序打印出了消息，几乎都一样，可是我们明白这是模拟出来的，并且是并行的方式。让 main()函数暂停 10 秒从而确定它会在另外两个协程之后结束。如果不这样（如果让 main()函数停止 4 秒），main()会提前结束，longWait()则无法完成。如果不在 main()中等待，协程会随着程序的结束而消亡。

当 main()函数返回的时候，程序退出：它不会等待任何其他非 main 协程的结束。这就是为什么在服务器程序中，每一个请求都会启动一个协程来处理，server()函数必须保持运行状态。通常使用一个无限循环来达到这样的目的。

另外，协程是独立的处理单元，一旦陆续启动一些协程，你无法确定它们是什么时候

真正开始执行的。代码逻辑必须独立于协程调用的顺序。

为了对比使用一个线程连续调用的情况，移除 Go 语言关键字，重新运行程序：

```
/*
返回:
这里是 main()开始的地方
开始 longWait()
结束 longWait()
开始 shortWait()
结束 shortWait()
挂起 main()
这里是 main()结束的地方 // 过了 17 秒之后打印
*/
```

协程更有用的一个例子应该是在一个非常长的数组中查找一个元素。将数组分割为若干个不重复的切片，然后给每一个切片启动一个协程进行查找计算。这样许多并行的线程可以用于查找任务，整体的查找时间会缩短（除以协程的数量）。

10.2　协程（goroutine）

执行体是个抽象的概念，在操作系统层面有多个概念与之对应，比如操作系统自己掌管的进程（process）、进程内的线程（thread）以及进程内的协程（coroutine，也叫轻量级线程）。与传统的系统级线程和进程相比，协程的最大优势在于其“轻量级”，可以轻松创建上百万个协程而不会导致系统资源衰竭，而线程和进程通常最多也不能超过 1 万个。这也是协程也叫轻量级线程的原因。

多数语言在语法层面并不直接支持协程，而是通过库的方式支持，但用库的方式支持的功能也并不完整，比如仅仅提供轻量级线程的创建、销毁与切换等能力。如果在这样的轻量级线程中调用一个同步 I/O 操作，比如网络通信、本地文件读写，都会阻塞其他的并发执行的轻量级线程，从而无法真正达到轻量级线程本身期望达到的目标。

Go 语言在语言级别支持轻量级线程，叫 goroutine。Go 语言标准库提供的所有系统调用操作（当然也包括所有同步 I/O 操作），都会出让 CPU 给其他 goroutine（创建协程会自动分配一个合适的 CPU 优先级，不管优先级如何都会与同级协程竞争 CPU 资源，从外部看来就是出让了部分 CPU 资源）。这让轻量级线程的切换管理不依赖于系统的线程和进程，也不依赖于 CPU 的核心数量，而是交给 Go 语言运行时负责统一调度（当然也允许手动控制）。

10.2.1 协程基础

goroutine 是 Go 语言并行设计的核心。goroutine 说到底其实就是协程，但是它比线程更小，十几个 goroutine 可能体现在底层也就是五六个线程，Go 语言内部帮你实现了这些 goroutine 之间的内存共享。执行 goroutine 只需极少的栈内存（大概是 4KB～5KB），当然会根据相应的数据伸缩。也正因为如此，程序可同时运行成千上万个并发任务。goroutine 比 thread 更易用、更高效、更轻便。

goroutine 是通过 Go 程序的 runtime 管理的一个线程管理器。goroutine 通过 go 关键字实现了，其实就是一个普通的函数：

```
go hello(a, b, c)
```

通过关键字 go 就启动了一个 goroutine。下面来看一个例子：

```
// 示例代码 10-2
package main

import (
      "fmt"
      "runtime"
)

func say(s string) {
      for i := 0; i < 5; i++ {
            runtime.Gosched()
            fmt.Println(s)
      }
}

func main() {
      go say("world") // 开一个新的 goroutine 执行
      say("hello") // 当前 goroutine 执行
}
/*
返回:
hello
world
hello
world
hello
world
hello
world
hello
*/
```

可以看到 go 关键字很方便地就实现了并发编程。上面的多个 goroutine 运行在同一个进程里面，共享内存数据，不过设计上要遵循：不要通过共享来通信，而要通过通信来共享。

runtime.Gosched()表示让 CPU 把时间片让给别人，下次某个时候继续恢复执行该 goroutine。在默认情况下，在 Go 1.5 以后将标识并发系统线程个数的 runtime.GOMAXPROCS 的初始值由 1 改为了运行环境的 CPU 核数。

可以通过设置 runtime.GOMAXPROCS(n)告诉调度器最大可以使用多少个线程，GOMAXPROCS 设置了同时运行逻辑代码的系统线程的最大数量，并返回之前的设置。如果 n < 1，不会改变当前设置。

10.2.2　协程间通信

从上面的例子可以看到，关键字 go 的引入使得在 Go 语言中并发编程变得简单而便捷，但同时也应该意识到并发编程的原生复杂性，并时刻对并发中容易出现的问题保持警惕。别忘了，我们的例子还不能正常工作呢。

事实上，不管是什么平台，什么编程语言，不管在哪里，并发都是一个大话题，话题大小通常也直接对应于问题的大小。并发编程的难度在于协调，而协调就要通过交流，从这个角度看来，并发单元间的通信是最大的问题。

通常有两种最常见的并发通信模型：共享数据和消息。

共享数据是指多个并发单元分别保持对同一个数据的引用，实现对该数据的共享。被共享的数据可能有多种形式，比如内存数据块、磁盘文件、网络数据等。在实际工程应用中最常见的无疑是内存了，也就是常说的共享内存。

将代码修改为如下形式：

```
// 示例代码 10-3
package main

import (
    "fmt"
    "runtime"
    "sync"
)

var counter int = 0

func Count(lock *sync.Mutex) {
```

```
        lock.Lock()
        counter++
        fmt.Println(z)
        lock.Unlock()
}
func main() {
        lock := &sync.Mutex{}
        for i := 0; i < 10; i++ {
                go Count(lock)
        }
        for {
                lock.Lock()
                c := counter
                lock.Unlock()
                runtime.Gosched()
                if c >= 10 {
                        break
                }
        }
}
```

此时这个例子终于可以正常工作了。在上面的例子中，在 10 个 goroutine 中共享了变量 counter。每个 goroutine 执行完成后，将 counter 的值加 1。因为 10 个 goroutine 是并发执行的，所以还引入了锁，也就是代码中的 lock 变量。每次对 n 的操作，都要先将锁锁住，操作完成后，再将锁打开。在主函数中，使用 for 循环来不断检查 counter 的值（同样需要加锁）。当其值达到 10 时，说明所有 goroutine 都执行完毕了，这时主函数返回，程序退出。

但是实现一个如此简单的功能，却写出如此臃肿而且难以理解的代码。想象一下，在一个大的系统中具有无数的锁、无数的共享变量、无数的业务逻辑与错误处理分支，那将是一场噩梦（目前主流语言基本都有这个问题）。Go 语言既然以并发编程作为语言的核心优势，当然不至于将这样的问题用这么无奈的方式来解决。Go 语言提供的是另一种通信模型，即以消息机制而非共享内存作为通信方式。

消息机制认为每个并发单元是自包含的、独立的个体，并且都有自己的变量，但在不同并发单元间这些变量不共享。每个并发单元的输入和输出只有一种，那就是消息。这有点类似于进程的概念，每个进程不会被其他进程打扰，它只做好自己的工作就可以了。不同进程间靠消息来通信，它们不会共享内存。

Go 语言提供的消息通信机制被称为通道（channel），接下来将详细介绍 channel。

最后再强调一遍，不要通过共享内存来通信，而应该通过通信来共享内存。

10.3　通道（channel）

channel 是 Go 语言在语言级别提供的 goroutine 间的通信方式，可以使用 channel 在两个或多个 goroutine 之间传递消息。channel 是进程内的通信方式，因此通过 channel 传递对象的过程和调用函数时的参数传递行为比较一致，比如也可以传递指针等。如果需要跨进程通信，建议用分布式系统的方法来解决，比如使用 Socket 或者 HTTP 等通信协议。Go 语言对于网络方面也有非常完善的支持。

channel 是类型相关的，也就是说，一个 channel 只能传递一种类型的值，这个类型需要在声明 channel 时指定。如果对 UNIX 管道有所了解的话，就不难理解 channel，可以将其认为是一种类型安全的管道。

在了解 channel 的语法前，先看一下用 channel 的方式重写上面的例子是什么样子的，以此对 channel 先有一个直观的认识，如下面代码所示：

```
// 示例代码 10-4
package main

import "fmt"

func Count(ch chan int) {
      ch <- 1
      fmt.Println("Counting")
}
func main() {
      chs := make([]chan int, 10)
      for i := 0; i < 10; i++ {
            chs[i] = make(chan int)
            go Count(chs[i])
      }
      for _, ch := range chs {
            <-ch
      }
}
```

在这个例子中，定义了一个包含 10 个 channel 的数组（名为 chs），并把数组中的每个 channel 分配给 10 个不同的 goroutine。在每个 goroutine 的 Add()函数完成后，通过 ch <- 1 语句向对应的 channel 中写入一个数据。在这个 channel 被读取前，这个操作是阻塞的。在所有的 goroutine 启动完成后，通过<-ch 语句从 10 个 channel 中依次读取数据。在对应的 channel 写入数据前，这个操作也是阻塞的。这样，就用 channel 实现了类似锁的功能，

仅供非商业用途或交流学习使用

进而保证了所有 goroutine 完成后主函数才返回。

在使用 Go 语言开发时，经常会遇到需要实现条件等待的场景，这也是 channel 可以发挥作用的地方。对 channel 的熟练使用，才能真正理解和掌握 Go 语言并发编程。下面学习 channel 的基本语法。

10.3.1 基本语法

一般 channel 的声明形式为：

```
var chanName chan ElementType
```

与一般的变量声明不同的地方仅仅是在类型之前加了 chan 关键字。ElementType 指定这个 channel 所能传递的元素类型。举个例子，声明一个传递类型为 int 的 channel：

```
var ch chan int
```

或者，声明一个 map，元素是 bool 型的 channel：

```
var m map[string] chan bool
```

上面的语句都是合法的。定义一个 channel 也很简单，直接使用内置的函数 make()即可：

```
ch := make(chan int)
```

这就声明并初始化了一个 int 型的名为 ch 的 channel。在 channel 的用法中，最常见的包括写入和读出。将一个数据写入（发送）至 channel 的语法很直观：

```
ch <- value
```

向 channel 写入数据通常会导致程序阻塞，直到有其他 goroutine 从这个 channel 中读取数据。从 channel 中读取数据的语法是：

```
value := <-ch
```

如果 channel 之前没有写入数据，那么从 channel 中读取数据也会导致程序阻塞，直到 channel 中被写入数据为止。之后还会介绍如何控制 channel 只接受写或者只允许读，即单向 channel。

10.3.2 select

早在 UNIX 时代，select 机制就已经被引入。通过调用 select()函数来监控一系列的文件句柄，一旦其中一个文件句柄发生了 I/O 动作，该 select()调用就会被返回。后来该机制

仅供非商业用途或交流学习使用

也被用于实现高并发的 Socket 服务器程序。Go 语言直接在语言级别支持 select 关键字，用于处理异步 I/O 问题。

select 的用法与 switch 语言非常类似，由 select 开始一个新的选择块，每个选择条件由 case 语句来描述。与 switch 语句可以选择任何可使用相等比较的条件相比，select 有比较多的限制，其中最大的一条限制就是每个 case 语句里必须是一个 I/O 操作，大致的结构如下：

```
select {
    case <-chan1: // 如果 chan1 成功读到数据，则进行该 case 处理语句
    case chan2 <- 1: // 如果成功向 chan2 写入数据，则进行该 case 处理语句
    default: // 如果上面都没有成功，则进入 default 处理流程
}
```

可以看出，select 不像 switch，后面并不带判断条件，而是直接去查看 case 语句。每个 case 语句都必须是一个面向 channel 的操作。比如上面的例子中，第一个 case 试图从 chan1 读取一个数据并直接忽略读到的数据，而第二个 case 则是试图向 chan2 中写入一个整数 1，如果这两者都没有成功，则到达 default 语句。

基于此功能，可以实现一个有趣的程序：

```
ch := make(chan int, 1)
for {
    select {
        case ch <- 0:
        case ch <- 1:
    }
    i := <-ch
    fmt.Println("接收到的值为：", i)
}
```

这个程序实现了一个随机向 ch 写入一个 0 或者 1 的过程。当然，这是个死循环。

10.3.3　缓冲机制

之前示范创建的都是不带缓冲的 channel，这种做法对于传递单个数据的场景可以接受，但对于需要持续传输大量数据的场景就有些不合适了。接下来介绍如何给 channel 带上缓冲，从而达到消息队列的效果。

要创建一个带缓冲的 channel，其实也非常容易：

```
c := make(chan int, 1024)
```

仅供非商业用途或交流学习使用

在调用 make()时将缓冲区大小作为第二个参数传入即可，比如上面这个例子就创建了一个大小为 1024 的 int 类型 channel，即使没有读取方，写入方也可以一直往 channel 里写，在缓冲区被填完之前都不会阻塞。

从带缓冲的 channel 中读取数据可以使用与常规非缓冲 channel 完全一致的方法，但也可以使用 range 关键字实现更为简便的循环读取：

```
for i := range c {
      fmt.Println("接收到的值为：", i)
}
```

10.3.4 超时和计时器

在之前对 channel 的介绍中，完全没有提到错误处理的问题，而这个问题显然是不能被忽略的。在并发编程的通信过程中，最需要处理的就是超时问题，即向 channel 写数据时发现 channel 已满，或者从 channel 试图读取数据时发现 channel 为空。如果不正确处理这些情况，很可能会导致整个 goroutine 锁死。

虽然 goroutine 是 Go 语言引入的新概念，但通信锁死问题已经存在很长时间了，在之前的 C/C++开发中也存在。操作系统在提供此类系统级通信函数时也会考虑超时的场景，因此这些方法通常都会带一个独立的超时参数。超过设定的时间时，仍然没有处理完任务，则该方法会立即终止并返回对应的超时信息。超时机制本身虽然也会带来一些问题，比如在运行比较快的机器或者高速的网络上运行正常的程序，到了慢速的机器或者网络上运行就会出问题，从而出现结果不一致的现象，但从根本上来说，解决死锁问题的价值要远大于所带来的问题。

使用 channel 时需要小心，比如对于以下这个用法：

```
i := <-ch
```

不出问题的话一切都正常运行，但如果出现了一个错误情况，即永远都没有人往 ch 里写数据，那么上述这个读取动作也将永远无法从 ch 中读取到数据，导致的结果就是整个 goroutine 永远阻塞并没有挽回的机会。如果 channel 只是被同一个开发者使用，那样出问题的可能性还低一些。但如果一旦对外公开，就必须考虑到最差的情况并对程序进行保护。

Go 语言没有提供直接的超时处理机制，但可以利用 select 机制。虽然 select 机制不是专为超时而设计的，却能很方便地解决超时问题，因为 select 的特点是只要其中一个 case 已经完成，程序就会继续往下执行，而不会考虑其他 case 的情况。

仅供非商业用途或交流学习使用

基于此特性，来为 channel 实现超时机制：

```
// 首先实现并执行一个匿名的超时等待函数
timeout := make(chan bool, 1)
go func() {
      time.Sleep(1e9) // 等待 1 秒
      timeout <- true
      }()
      // 然后把 timeout 这个 channel 利用起来
      select {
      case <-ch:
            // 从 ch 中读取到数据
      case <-timeout:
            // 一直没有从 ch 中读取到数据，但从 timeout 中读取到了数据
}
```

10.3.5　channel 的传递

需要注意的是，在 Go 语言中 channel 本身也是一个原生类型，与 map 之类的类型地位一样，因此 channel 本身在定义后也可以通过 channel 来传递。

可以使用这个特性来实现 Linux/UNIX 非常常见的管道（pipe）特性。管道也是使用非常广泛的一种设计模式，比如在处理数据时，可以采用管道设计，这样可以比较容易以插件的方式增加数据的处理流程。

下面利用 channel 可被传递的特性来实现管道。为了简化表达，假设在管道中传递的数据只是一个整型数，在实际的应用场景中通常会是一个数据块。

首先限定基本的数据结构：

```
type PipeData struct {
      value   int
      handler func(int) int
      next    chan int
}
```

然后写一个常规的处理函数。只要定义一系列 PipeData 的数据结构并一起传递给这个函数，就可以达到流式处理数据的目的：

```
func handle(queue chan *PipeData) {
      for data := range queue {
            data.next <- data.handler(data.value)
      }
}
```

仅供非商业用途或交流学习使用

这里只给出了大概的样子，限于篇幅不再展开。同理，利用 channel 这个可传递特性，可以实现非常强大、灵活的系统架构。相比之下，在 C++、Java、C#中，要达成这样的效果，通常就意味着要设计一系列接口。

与 Go 语言接口的非侵入式类似，channel 的这些特性也可以大大减少开发时间，用一些比较简单却实用的方式来达成在其他语言中需要使用众多技巧才能达成的效果。

10.3.6 单向 channel

顾名思义，单向 channel 只能用于发送或者接收数据。channel 本身必然是同时支持读写的，否则根本没法用。假如一个 channel 真的只能读，那么肯定只会是空的，因为你没机会往里面写数据。同理，如果一个 channel 只允许写，即使写进去了，也没有丝毫意义，因为没有机会读取里面的数据。所谓的单向 channel 概念，其实只是对 channel 的一种使用限制。

在将一个 channel 变量传递到一个函数时，可以通过将其指定为单向 channel 变量，从而限制该函数中可以对此 channel 的操作，比如只能往这个 channel 写，或者只能从这个 channel 读。

单向 channel 变量的声明非常简单，例如：

```
var ch1 chan int // ch1是一个正常的channel，不是单向的
var ch2 chan<- float64 // ch2是单向channel，只用于写float64数据
var ch3 <-chan int // ch3是单向channel，只用于读取int数据
```

关于单向 channel 的初始化，之前已经提到过，channel 是一个原生类型，因此不仅支持被传递，还支持类型转换。只有在介绍了单向 channel 的概念后，读者才会明白类型转换对于 channel 的意义：在单向 channel 和双向 channel 之间进行转换。示例如下：

```
ch4 := make(chan int)
ch5 := <-chan int(ch4) // ch5就是一个单向的读取channel
ch6 := chan<- int(ch4) // ch6是一个单向的写入channel
```

基于 ch4，通过类型转换初始化了两个单向 channel：单向读的 ch5 和单向写的 ch6。

为什么会这样限制呢？从设计的角度考虑，所有的代码应该都遵循“最小权限原则”，从而避免没必要的使用泛滥问题，进而导致程序失控。写过 C++程序的读者肯定就会联想起 const 指针的用法。非 const 指针具备 const 指针的所有功能，将一个指针设定为 const 就是明确告诉函数实现者不要试图对该指针进行修改。单向 channel 也是起到这样的一种契约作用。

仅供非商业用途或交流学习使用

我们来看看单向 channel 的用法：

```
func Parse(ch <-chan int) {
      for value := range ch {
            fmt.Println("解析后的值为：", value)
      }
}
```

除非这个函数的实现者使用了类型转换，否则这个函数就不会因为各种原因而对 ch 进行写，避免在 ch 中出现非期望的数据，从而很好地实践最小权限原则。

10.3.7　关闭 channel

关闭 channel 非常简单，直接使用 Go 语言内置的 close()函数即可：

```
close(ch)
```

在介绍了如何关闭 channel 之后，就多了一个问题：如何判断一个 channel 是否已经被关闭？

可以在读取的时候使用多重返回值的方式：

```
x, ok := <-ch
```

这个用法与 map 中的按键获取 value 的过程比较类似，只需要看第二个 bool 返回值即可，如果返回值是 false 则表示 ch 已经被关闭。

10.4　并发进阶

10.4.1　多核并行化

在执行一些“昂贵”的计算任务时，我们希望能够尽量利用现代服务器普遍具备的多核特性尽量将任务并行化，从而达到降低总计算时间的目的。此时需要了解 CPU 核心的数量，并有针对性地分解计算任务到多个 goroutine 中去并行运行。

下面模拟一个完全可以并行的计算任务，计算 n 个整型数的总和。可以将所有整型数分成 m 份，m 即 CPU 的个数。让每个 CPU 开始计算分给它的那份计算任务，最后将每个 CPU 的计算结果再做一次累加，这样就可以得到所有 n 个整型数的总和：

```
type Vector []float64

// 分配给每个 CPU 的计算任务
```

仅供非商业用途或交流学习使用


```
func (v Vector) DoSome(i, n int, u Vector, c chan int) {
	for ; i < n; i++ {
		v[i] += u.Op(v[i])
	}
	c <- 1 // 发信号告诉任务管理者我已经计算完成了
}

const NCPU = 8 // 假设总共有 8 核
func (v Vector) DoAll(u Vector) {
	c := make(chan int, NCPU) // 用于接收每个 CPU 的任务完成信号
	for i := 0; i < NCPU; i++ {
		go v.DoSome(i*len(v)/NCPU, (i+1)*len(v)/NCPU, u, c)
	}

	// 等待所有 CPU 的任务完成
	for i := 0; i < NCPU; i++ {
		<-c // 获取到一个数据，表示一个 CPU 计算完成了
	}
	// 到这里表示所有计算已经结束
}
```

这两个函数看起来设计非常合理。DoAll()会根据 CPU 核心的数目对任务进行分割，然后开辟多个 goroutine 并行执行这些计算任务。

是否可以将总的计算时间降到接近原来的 1/n 呢？答案是不一定。如果使用 Benchmark 方法，会发现总的执行时间没有明显缩短。再去观察 CPU 运行状态，你会发现尽管有 8 个 CPU 核心，但在计算过程中其实只有一个 CPU 核心处于繁忙状态，这是会让很多 Go 语言初学者迷惑的问题。

官方的答案是，这是当前版本的 Go 语言编译器还不能很智能地去发现和利用多核的优势。虽然确实创建了多个 goroutine，并且从运行状态看这些 goroutine 也都在并行运行，但实际上所有这些 goroutine 都运行在同一个 CPU 核心上，在一个 goroutine 得到时间片执行的时候，其他 goroutine 都会处于等待状态。从这一点可以看出，虽然 goroutine 简化了写并行代码的过程，但实际上整体运行效率并不真正高于单线程程序。

在 Go 语言升级到默认支持多 CPU 的某个版本之前，可以先通过设置环境变量 GOMAXPROCS 的值来控制使用多少个 CPU 核心。具体操作方法是通过直接设置环境变量 GOMAXPROCS 的值，或者在代码中启动 goroutine 之前先调用以下这个语句以设置使用 8 个 CPU 核心：

```
runtime.GOMAXPROCS(16)
```

到底应该设置多少个CPU核心呢？其实runtime包中还提供了另外一个函数NumCPU()

来获取核心数。可以看到，Go 语言其实已经感知到所有的环境信息，下一版本中完全可以利用这些信息将 goroutine 调度到所有 CPU 核心上，从而最大化地利用服务器的多核计算能力。抛弃 GOMAXPROCS 只是个时间问题。

10.4.2　协程同步

通道可以被关闭，尽管它们和文件不同，不必每次都关闭，只有在当需要告诉接收者不会再提供新的值的时候，才需要关闭通道。只有发送者需要关闭通道，接收者永远不会需要。继续看示例：

```
// 示例代码 10-5
package main

import (
    "fmt"
    "time"
)

func main() {
    ch := make(chan string)

    go sendData(ch)
    go getData(ch)

    time.Sleep(1e9)
}

func sendData(ch chan string) {
    ch <- "纽约"
    ch <- "华盛顿"
    ch <- "伦敦"
    ch <- "北京"
    ch <- "东京"
}

func getData(ch chan string) {
    var input string
    // time.Sleep(1e9)
    for {
        input = <-ch
        fmt.Printf("%s ", input)
    }
}
/*
返回：
```

```
纽约 华盛顿 伦敦 北京 东京
*/
```

如何在通道的 sendData()完成的时候发送一个信号，getData()又如何检测到通道是否关闭或阻塞？

第一种方法可以通过函数 close(ch)来完成：这个将通道标记为无法通过发送操作<-接收更多的值；给已经关闭的通道发送或者再次关闭都会导致运行时的 panic。在创建一个通道后使用 defer 语句是个不错的办法（类似这种情况）：

```
ch := make(chan float64)
defer close(ch)
```

第二种方法可以使用逗号、ok 操作符：用来检测通道是否被关闭：

```
v, ok := <-ch   // 如果 v 接收到一个值，ok 的值则为 true
```

通常和 if 语句一起使用：

```
if v, ok := <-ch; ok {
  process(v)
}
```

或者在 for 循环中接收的时候，当关闭或者阻塞的时候使用 break：

```
v, ok := <-ch
if !ok {
  break
}
process(v)
```

可以通过_ = ch <- v 来实现非阻塞发送，因为空标识符获取到了发送给 ch 的所有内容。

实现非阻塞通道的读取，需要使用 select：

```
// 示例代码 10-6
package main

import "fmt"

func main() {
    ch := make(chan string)
    go sendData(ch)
    getData(ch)
}

func sendData(ch chan string) {
    ch <- "纽约"
    ch <- "华盛顿"
    ch <- "伦敦"
```

```
    ch <- "北京"
    ch <- "东京"
    close(ch)
}

func getData(ch chan string) {
    for {
        input, open := <-ch
        if !open {
            break
        }
        fmt.Printf("%s ", input)
    }
}
```

改变了以下代码：

（1）现在只有 sendData()是协程，getData()和 main()在同一个线程中。

```
go sendData(ch)
getData(ch)
```

（2）在 sendData()函数的最后，关闭了通道：

```
func sendData(ch chan string) {
    ch <- "纽约"
    ch <- "华盛顿"
    ch <- "伦敦"
    ch <- "北京"
    ch <- "东京"
    close(ch)
}
```

（3）在 for 循环的 getData()中，在每次接收通道的数据之前都使用 if !open 来检测：

```
for {
  input, open := <-ch
  if !open {
    break
  }
  fmt.Printf("%s ", input)
}
```

使用 for-range 语句来读取通道是更好的办法，因为这会自动检测通道是否关闭：

```
for input := range ch {
  process(input)
}
```

阻塞和生产者—消费者模式：在通道迭代器中，两个协程经常是一个阻塞另外一个。

如果程序工作在多核的机器上，大部分时间只用到一个处理器。可以通过使用带缓冲（缓冲空间大于 0）的通道来改善。比如，缓冲大小为 100，迭代器在阻塞之前，至少可以从容器获得 100 个元素。如果消费者协程在独立的内核运行，就有可能让协程不会出现阻塞。

由于容器中元素的数量通常是已知的，需要让通道有足够的容量放置所有的元素。这样，迭代器就不会阻塞（尽管消费者协程仍然可能阻塞）。然后，这样有效地加倍了迭代容器所需要的内存使用量，所以通道的容量需要限制最大值。记录运行时间和性能测试可以帮助你找到最小的缓存容量带来最好的性能。

10.4.3 协程和恢复

在下面的代码中，如果 do(work)发生 panic，错误会被记录且协程会退出并释放，而其他协程不受影响：停掉了服务器内部一个失败的协程而不影响其他协程的工作。

```
func server(workChan <-chan *Work) {
    for work := range workChan {
        go safelyDo(work)   // goroutine 开始工作
    }
}

func safelyDo(work *Work) {
    defer func {
        if err := recover(); err != nil {
            log.Printf("%s 在%v 中执行失败", err, work)
        }
    }()
    do(work)
}
```

因为 recover 总是返回 nil，除非直接在 defer 修饰的函数中调用，defer 修饰的代码可以调用那些自身可以使用 panic 和 recover 避免失败的库例程（库函数）。

例如，safelyDo()中 deffer 修饰的函数可能在调用 recover 之前就调用了一个 logging 函数，panicking 状态不会影响 logging 代码的运行。

因为加入了恢复模式，函数 do（以及它调用的任何对象）可以通过调用 panic 来摆脱不好的情况。但是恢复是在 panic 的协程内部的，它不能被另外一个协程恢复。

10.5　本章小结

Go 语言运行时的调度器是一个复杂的软件，能管理被创建的所有 goroutine 并为其分配执行时间。这个调度器在操作系统之上，将操作系统的线程与语言运行时的逻辑处理器绑定，并在逻辑处理器上运行 goroutine。调度器在任何给定的时间，都会全面控制哪个 goroutine 要在哪个逻辑处理器上运行。

Go 语言的并发同步模型来自一个叫作通信顺序进程（Communicating Sequential Processes，CSP）的范型（paradigm）。CSP 是一种消息传递模型，通过在 goroutine 之间传递数据来传递消息，而不是对数据进行加锁来实现同步访问。

第 11 章
网络编程

Go 语言追求的是性能、简单，本章将介绍如何使用 Socket 编程，很多游戏服务都是采用 Socket 来编写服务器端的，因为 HTTP 协议相对而言比较耗费性能，本章会学习如何使用 Go 语言来进行 Socket 编程。此外作为最常见的 Web 服务，HTTP 编程自然也不能缺席；还有数据库、Cookie 操作等也是网络编程中的一个重镇，本章都会一一讲解。

11.1 Socket 编程

对于底层网络应用开发者而言，几乎所有网络编程都是 Socket，因为大部分底层网络的编程都离不开 Socket 编程。HTTP 编程、Web 开发、IM 通信、视频流传输的底层都是 Socket 编程。

日常生活中我们每天打开浏览器浏览网页、使用 QQ 聊天、邮件收发、直播等，客户端和服务器端的通信在底层看来都是依靠 Socket 通信的，可见 Socket 编程在现代编程中占据了多么重要的地位，这一节将介绍 Go 语言中如何进行 Socket 编程。

11.1.1 什么是 Socket

Socket 起源于 UNIX，而 UNIX 的基本哲学之一就是“一切皆文件”，都可以用“打开（open）→读写（write/read）→关闭（close）”模式来操作，Socket 就是该模式的一个实现，网络的 Socket 数据传输是一种特殊的 I/O，Socket 也是一种文件描述符。Socket 也具有一个类似于打开文件的函数调用：Socket()，该函数返回一个整型的 Socket 描述符，随后的连接建立、数据传输等操作都是通过该 Socket 实现的。

严格意义来说，本节介绍的 Socket 编程主要是面向 OSI 模型的第三层和第四层协议，再具体点就是主要针对 IP 协议、TCP 协议和 UDP 协议，而 HTTP 这种基于 TCP 的七层协议在下一节会有介绍。所以本节介绍的 Socket 类型只有两种：流式 Socket（SOCK_STREAM）和数据报式 Socket（SOCK_DGRAM）。流式 Socket 是一种面向连接的 Socket，针对面向连接的 TCP 服务应用；数据报式 Socket 是一种无连接的 Socket，对应无连接的 UDP 服务应用。

1. Socket 如何通信

网络中的进程之间如何通过 Socket 通信呢？首要解决的问题是如何唯一标识一个进程，否则通信无从谈起！在本地可以通过进程 PID 来唯一标识一个进程，但是在网络中这是行不通的。其实 TCP/IP 协议族已经帮我们解决了这个问题，网络层的“IP 地址”可以唯一标识网络中的主机，而传输层的“协议+端口”可以唯一标识主机中的应用程序（进程）。这样利用三大要素（IP 地址、协议、端口）就可以标识网络的进程了，网络中需要互相通信的进程，就可以利用这个标志在它们之间进行交互。

使用 TCP/IP 协议的应用程序通常采用应用编程接口：UNIX BSD 的套接字和 UNIX System V 的 TLI（已经被淘汰），来实现网络进程之间的通信。就目前而言，几乎所有的应用程序都是采用 Socket，而现在又是网络时代，网络中进程通信是无处不在的，这就是为什么说“一切皆 Socket”。通过上面的介绍我们知道 Socket 有两种：TCP Socket 和 UDP Socket，TCP 和 UDP 是协议，而要确定一个进程得需要三要素，所以还需要 IP 地址和端口。

2. Go 语言支持的 IP 类型

在 Go 语言的 net 包中定义了很多类型、函数和方法用于网络编程，其中 IP 的定义如下：

```
type IP []byte
```

在 net 包中有很多函数用于操作 IP，但是其中比较常用的也就几个，其中 ParseIP(s string) IP 函数会把一个 IPv4 或者 IPv6 的地址转化成 IP 类型，请看下面的例子:

```
// 示例代码 11-1
package main
import (
      "net"
      "os"
      "fmt"
)
func main() {
```

```
        if len(os.Args) != 2 {
                fmt.Fprintf(os.Stderr, "用法: %s IP 地址\n", os.Args[0])
                os.Exit(1)
        }
        name := os.Args[1]
        addr := net.ParseIP(name)
        if addr == nil {
                fmt.Println("无效的 IP 地址")
        } else {
                fmt.Println("IP 地址是", addr.String())
        }
        os.Exit(0)
}
```

执行之后就会发现只要输入一个 IP 地址就会给出相应的 IP 格式。

11.1.2 Dial()函数

在 Go 语言中编写网络程序时，将看不到传统的编码形式。Go 语言中 Socket 编程的 API 都在 net 包中。Go 语言提供了 Dial 函数来连接服务器，使用 Listen 监听，Accept 接收连接，所以 Go 语言的网络编程和其他同类语言（C 语言）一样有着相似的 API，只需了解即可。

Go 语言标准库对传统的 Socket 编程过程进行了抽象和封装。无论期望使用什么协议建立什么形式的连接，都只需要调用 net.Dial()即可。

Dial()函数的原型如下：

```
func Dial(net, addr string) (Conn, error)
```

其中 net 参数是网络协议的名字，addr 参数是 IP 地址或域名，而端口号以“:”的形式跟随在地址或域名的后面，端口号可选。如果连接成功，返回连接对象，否则返回 error。

下面来看看几种常见协议的调用方式。

TCP 链接：

```
conn, err := net.Dial("tcp", "192.168.0.10:2100")
```

UDP 链接：

```
conn, err := net.Dial("udp", "192.168.0.12:975")
```

ICMP 链接（使用协议名称）：

```
conn, err := net.Dial("ip4:icmp", "www.baidu.com")
```

ICMP 链接（使用协议编号）：

```
conn, err := net.Dial("ip4:1", "10.0.0.3")
```

这里可以通过以下链接查看协议编号的含义：

http://www.iana.org/assignments/protocol-num-bers/protocol-numbers.xml

目前，Dial()函数支持如下几种网络协议：TCP、TCP4（仅限 IPv4）、TCP6（仅限 IPv6）、UDP、UDP4（仅限 IPv4）、UDP6（仅限 IPv6）、IP、IP4（仅限 IPv4）和 IP6（仅限 IPv6）。

在成功建立连接后，就可以进行数据的发送和接收。在发送数据时，使用 conn 的 Write()成员方法，在接收数据时使用 Read()方法。

11.1.3　ICMP 示例

下面实现这样一个例子：使用 ICMP 协议向在线的主机发送一个问候，并等待主机返回，具体代码如下所示：

```
// 示例代码 11-2
package main

import (
	"bytes"
	"fmt"
	"io"
	"net"
	"os"
)

func main() {
	if len(os.Args) != 2 {
		fmt.Println("用法: ", os.Args[0], "hostname")
		os.Exit(1)
	}
	service := os.Args[1]

	conn, err := net.Dial("ip4:icmp", service)
	checkError(err)

	var msg [512]byte
	msg[0] = 8  // echo
	msg[1] = 0  // code 0
	msg[2] = 0  // checksum
	msg[3] = 0  // checksum
	msg[4] = 0  // identifier[0]
```

```
        msg[5] = 13 // identifier[1]
        msg[6] = 0  // sequence[0]
        msg[7] = 37 // sequence[1]
        len := 8
        check := checkSum(msg[0:len])
        msg[2] = byte(check >> 8)
        msg[3] = byte(check & 255)

        _, err = conn.Write(msg[0:len])
        checkError(err)

        _, err = conn.Read(msg[0:])
        checkError(err)

        fmt.Println("取得响应")
        if msg[5] == 13 {
              fmt.Println("Identifier 匹配")
        }
        if msg[7] == 37 {
              fmt.Println("Sequence 匹配")
        }

        os.Exit(0)
  }
  func checkSum(msg []byte) uint16 {
        sum := 0

        // 先假设为偶数
        for n := 1; n < len(msg)-1; n += 2 {
              sum += int(msg[n])*256 + int(msg[n+1])
        }
        sum = (sum >> 16) + (sum & 0xffff)
        sum += (sum >> 16)
        var answer uint16 = uint16(^sum)
        return answer
  }
  func checkError(err error) {
        if err != nil {
              fmt.Fprintf(os.Stderr, "错误: %s", err.Error())
              os.Exit(1)
        }
  }
  func readFully(conn net.Conn) ([]byte, error) {
        defer conn.Close()
        result := bytes.NewBuffer(nil)
        var buf [512]byte
        for {
              n, err := conn.Read(buf[0:])
```

```
            result.Write(buf[0:n])
            if err != nil {
                if err == io.EOF {
                        break
                }
                return nil, err
            }
        }
        return result.Bytes(), nil
}
/*
返回:
./icmpTest www.baidu.com
取得响应
Identifier 匹配
Sequence 匹配
*/
```

11.1.4　TCP Socket

当知道如何通过网络端口访问一个服务时能够做什么呢？作为客户端来说，可以通过向远端某台机器的某个网络端口发送一个请求，然后得到在机器的此端口上监听的服务反馈的信息。作为服务器端，需要把服务绑定到某个指定端口，并且在此端口上监听，当有客户端来访问时能够读取信息并且写入反馈信息。

在 Go 语言的 net 包中有一个类型 TCPConn，这个类型可以用来作为客户端和服务器端交互的通道，它有两个主要的函数：

```
func (c *TCPConn) Write(b []byte) (n int, err os.Error)
func (c *TCPConn) Read(b []byte) (n int, err os.Error)
```

TCPConn 可以用在客户端和服务器端读写数据。

还需要知道一个 TCPAddr 类型，它表示一个 TCP 的地址信息，它的定义如下：

```
type TCPAddr struct {
      IP IP
      Port int

Zone string // IPv6 范围寻址区域
}
```

在 Go 语言中通过 ResolveTCPAddr 获取一个 TCPAddr：

```
func ResolveTCPAddr(net, addr string) (*TCPAddr, os.Error)
```

- net 参数是 TCP4、TCP6、TCP 中的任意一个，分别表示 TCP（IPv4-only），TCP

（IPv6-only）或者 TCP（IPv4，IPv6 的任意一个）。

- addr 表示域名或者 IP 地址，例如 www.google.com:80 或者 127.0.0.1:22。

1. TCP 客户端

Go 语言通过 net 包中的 DialTCP 函数来建立一个 TCP 连接，并返回一个 TCPConn 类型的对象，当连接建立时服务器端也创建一个同类型的对象，此时客户端和服务器端通过各自拥有的 TCPConn 对象进行数据交换。一般而言，客户端通过 TCPConn 对象将请求信息发送到服务器端，读取服务器端响应的信息。服务器端读取并解析来自客户端的请求，并返回应答信息，这个连接只有当任一端关闭连接之后才失效，不然该连接可以一直使用。建立连接的函数定义如下：

```
func DialTCP(net string, laddr, raddr *TCPAddr) (c *TCPConn, err os.Error)
```

- net 参数是 TCP4、TCP6、TCP 中的任意一个，分别表示 TCP（IPv4-only）、TCP（IPv6-only）或者 TCP（IPv4、IPv6 的任意一个）；
- laddr 表示本机地址，一般设置为 nil；
- raddr 表示远程服务地址。

接下来写一个简单的例子，模拟一个基于 HTTP 协议的客户端请求去连接一个 Web 服务器端。要写一个简单的 http 请求头，格式类似如下：

```
"HEAD / HTTP/1.0\r\n\r\n"
```

从服务器端接收到的响应信息格式可能如下：

```
HTTP/1.0 200 OK
ETag: "-9985996"
Last-Modified: Thu, 25 Mar 2017 17:51:10 GMT
Content-Length: 18074
Connection: close
Date: Sat, 28 Aug 2017 00:43:48 GMT
Server: lighttpd/1.4.23
```

客户端代码如下所示：

```
// 示例代码 11-3
package main

import (
      "fmt"
      "io/ioutil"
      "net"
      "os"
```

```
)

func main() {
	if len(os.Args) != 2 {
		fmt.Fprintf(os.Stderr, "用法: %s host:port ", os.Args[0])
		os.Exit(1)
	}
	service := os.Args[1]
	tcpAddr, err := net.ResolveTCPAddr("tcp4", service)
	checkError(err)
	conn, err := net.DialTCP("tcp", nil, tcpAddr)
	checkError(err)
	_, err = conn.Write([]byte("HEAD / HTTP/1.0\r\n\r\n"))
	checkError(err)
	result, err := ioutil.ReadAll(conn)
	checkError(err)
	fmt.Println(string(result))
	os.Exit(0)
}
func checkError(err error) {
	if err != nil {
		fmt.Fprintf(os.Stderr, "错误: %s", err.Error())
		os.Exit(1)
	}
}
```

通过上面的代码可以看出：首先程序将用户的输入作为参数 service 传入 net.ResolveTCPAddr 获取一个 tcpAddr，然后把 tcpAddr 传入 DialTCP 后创建了一个 TCP 连接 conn，通过 conn 来发送请求信息，最后通过 ioutil.ReadAll 从 conn 中读取全部的文本，也就是服务器端响应反馈的信息。

2. TCP 服务器端

上面编写了一个 TCP 的客户端程序，也可以通过 net 包来创建一个服务器端程序，在服务器端需要绑定服务到指定的非激活端口，并监听此端口，当有客户端请求到达的时候可以接收到来自客户端连接的请求。net 包中有相应功能的函数，函数定义如下：

```
func ListenTCP(net string, laddr *TCPAddr) (l *TCPListener, err os.Error)
func (l *TCPListener) Accept() (c Conn, err os.Error)
```

参数说明同 DialTCP 的参数一样。下面实现一个简单的时间同步服务，监听 8000 端口：

```
// 示例代码 11-4
package main
```

```
import (
	"fmt"
	"net"
	"os"
	"time"
)

func echo(conn *net.TCPConn) {
	tick := time.Tick(5 * time.Second) // 5s请求一次
	for now := range tick {
		n, err := conn.Write([]byte(now.String()))
		if err != nil {
			log.Println(err)
			conn.Close()
			return
		}
		fmt.Printf("send %d bytes to %s\n", n, conn.RemoteAddr())
	}
}
func main() {
	address := net.TCPAddr{
	IP:   net.ParseIP("127.0.0.1"),
	Port: 8000,
	}
	listener, err := net.ListenTCP("tcp4", &address) // 创建TCP4服务器端监听器
	if err != nil {
		log.Fatal(err) // Println + os.Exit(1)
	}
	for {
		conn, err := listener.AcceptTCP()
		if err != nil {
			log.Fatal(err) // 错误直接退出
		}
		fmt.Println("远程地址：", conn.RemoteAddr())
		go echo(conn)
	}
}
```

上面的服务“跑”起来之后，它将会一直在那里等待，直到有新的客户端请求到达。当有新的客户端请求到达并同意接受该请求的时候它会反馈当前的时间信息。值得注意的是，在代码中的 for 循环里，当有错误发生时，直接 continue 而不是退出，是因为在服务器端“跑”代码的时候，如果有错误发生，最好是由服务器端记录错误，然后当前连接的客户端直接报错而退出，从而不会影响到当前服务器端运行的整个服务。

以上代码还通过把业务处理分离到 echo 函数中，实现多并发执行连接。

在上面这个例子中所建立的连接是一个长连接，在客户端可以持续对服务器端发送数据并记录到服务器端的输出中，而保持长连接的关键即 conn.Read()，它会不断读取客户端发来的请求，但这不足以保持连接稳定。由于需要保持与客户端的长连接，所以不能在读取完一次请求后就关闭连接。由于 conn.SetReadDeadline()设置了超时，当一定时间内客户端无请求发送，conn 便会自动关闭，下面的 for 循环即会因为连接已关闭而跳出。需要注意的是，request 在创建时需要指定一个最大长度以防止洪水攻击（flood attack），由于 conn.Read()会将新读取到的内容连接到原内容之后，所以如果清除之前的信息需要在每次请求读取之后清理 request，例如：request = make([]byte, 128)，限制连接大小最大值为 128B。

3. 控制 TCP 连接

TCP 有很多连接控制函数，平常用到比较多的有如下几个函数：

```
func DialTimeout(net, addr string, timeout time.Duration) (Conn, error)
```

设置建立连接的超时时间，客户端和服务器端都适用，当超过设置时间后，连接自动关闭：

```
func (c *TCPConn) SetReadDeadline(t time.Time) error
func (c *TCPConn) SetWriteDeadline(t time.Time) error
```

用来设置写入/读取一个连接的超时时间。当超过设置时间时，连接自动关闭：

```
func (c *TCPConn) SetKeepAlive(keepalive bool) os.Error
```

设置 keepAlive 属性，是操作系统层在 TCP 上没有数据和 ACK 的时候，会间隔性地发送 keepalive 包，操作系统可以通过该包来判断一个 TCP 连接是否已经断开，在 Windows 上默认 2 小时没有收到数据和 keepalive 包的时候人为断开 TCP 连接，这个功能和通常在应用层加的心跳包的功能类似。

更多的内容请查看 net 包的文档。

11.1.5　UDP Socket

Go 语言包中处理 UDP Socket 和 TCP Socket 不同的地方就是在服务器端处理多个客户端请求数据包的方式不同，UDP 缺少了对客户端连接请求的 Accept 函数。其他几乎一模一样，只是 TCP 换成了 UDP 而已。UDP 的几个主要函数如下所示：

```
func ResolveUDPAddr(net, addr string) (*UDPAddr, os.Error)
func DialUDP(net string, laddr, raddr *UDPAddr) (c *UDPConn, err os.Error)
func ListenUDP(net string, laddr *UDPAddr) (c *UDPConn, err os.Error)
```

```
func (c *UDPConn) ReadFromUDP(b []byte) (n int, addr *UDPAddr, err os.Error)
func (c *UDPConn) WriteToUDP(b []byte, addr *UDPAddr) (n int, err os.Error)
```

一个 UDP 的客户端代码如下所示，可以看到不同的就是 TCP 换成了 UDP 而已：

```
// 示例代码 11-7
package main

import (
      "fmt"
      "net"
      "os"
)

func main() {
      if len(os.Args) != 2 {
            fmt.Fprintf(os.Stderr, "用法：%s host:port", os.Args[0])
            os.Exit(1)
      }
      service := os.Args[1]
      udpAddr, err := net.ResolveUDPAddr("udp4", service)
      checkError(err)
      conn, err := net.DialUDP("udp", nil, udpAddr)
      checkError(err)
      _, err = conn.Write([]byte("anything"))
      checkError(err)
      var buf [512]byte
      n, err := conn.Read(buf[0:])
      checkError(err)
      fmt.Println(string(buf[0:n]))
      os.Exit(0)
}
func checkError(err error) {
      if err != nil {
            fmt.Fprintf(os.Stderr, "错误：", err.Error())
            os.Exit(1)
      }
}
```

下面来看看 UDP 服务器端如何处理：

```
// 示例代码 11-8
package main

import (
      "fmt"
      "net"
      "os"
      "time"
```

```
)
func main() {
	service := ":1200"
	udpAddr, err := net.ResolveUDPAddr("udp4", service)
	checkError(err)
	conn, err := net.ListenUDP("udp", udpAddr)
	checkError(err)
	for {
		handleClient(conn)
	}
}
func handleClient(conn *net.UDPConn) {
	var buf [512]byte
	_, addr, err := conn.ReadFromUDP(buf[0:])
	if err != nil {
		return
	}
	daytime := time.Now().String()
	conn.WriteToUDP([]byte(daytime), addr)
}
func checkError(err error) {
	if err != nil {
		fmt.Fprintf(os.Stderr, "错误: ", err.Error())
		os.Exit(1)
	}
}
```

通过对 TCP 和 UDP Socket 编程的描述和实现，可见 Go 语言已经完备地支持了 Socket 编程，而且使用起来相当方便，Go 语言提供了很多函数，通过这些函数可以很容易就编写出高性能的 Socket 应用。

11.2　HTTP 编程

HTTP（HyperText Transfer Protocol，超文本传输协议）是互联网上应用最为广泛的一种网络协议，定义了客户端和服务器端之间请求与响应的传输标准。

Go 语言标准库内建提供了 net/http 包，涵盖了 HTTP 客户端和服务器端的具体实现。使用 net/http 包，可以很方便地编写 HTTP 客户端或服务器端的程序。

11.2.1　HTTP 客户端

Go 语言内置的 net/http 包提供了最简洁的 HTTP 客户端实现，无须借助第三方网络通

信库（比如 libcurl）就可以直接使用 HTTP 中用得最多的 GET 和 POST 方式请求数据。

1. 基本用法

net/http 包的 Client 类型提供了如下几个方法，利用最简洁的方式实现 HTTP 请求：

```
func (c *Client) Get(url string) (r *Response, err error)
func (c *Client) Post(url string, bodyType string, body io.Reader) (r *Response,
err error)
func (c *Client) PostForm(url string, data url.Values) (r *Response, err error)
func (c *Client) Head(url string) (r *Response, err error)
func (c *Client) Do(req *Request) (resp *Response, err error)
```

下面概要介绍这几个方法。

1）http.Get()

要请求一个资源，只需调用 http.Get()方法（等价于 http.DefaultClient.Get()）即可，示例代码如下：

```
resp, err := http.Get("http://example.com/")
      if err != nil {
      // 处理错误 ...
      return
}
defer resp.Body.close()
io.Copy(os.Stdout, resp.Body)
```

上面这段代码请求一个网站首页，并将其网页内容打印到标准输出流中。

2）http.Post()

要以 POST 的方式发送数据，也很简单，只需调用 http.Post()方法并依次传递这 3 个参数即可：请求的目标 URL、将要 POST 数据的资源类型（MIMEType）和数据的比特流（[]byte 形式）。

下面的示例代码演示了如何上传一张图片：

```
resp, err := http.Post("http://example.com/upload", "image/jpeg", &imageDataBuf)
if err != nil {
      // 处理错误
      return
}
if resp.StatusCode != http.StatusOK {
      // 处理错误
      return
}
// ...
```

3）http.PostForm()

http.PostForm()方法实现了标准编码格式为 application/x-www-form-urlencoded 的表单提交。下面的示例代码模拟 HTML 表单提交一篇新文章：

```
resp, err := http.PostForm("http://example.com/posts", {"article title"}, "content
": {"article body"}})
if err != nil {
    // 处理错误
    return
}
// ...
```

4）http.Head()

HTTP 中的 Head 请求方式表明只请求目标 URL 的头部信息，即 HTTP Header 而不返回 HTTP Body。Go 语言内置的 net/http 包同样也提供了 http.Head()方法，该方法同 http.Get()方法一样，只需传入目标 URL 一个参数即可。下面的示例代码请求一个网站首页的 HTTP Header 信息：

```
resp, err := http.Head("http://example.com/")
```

2. 封装

除了之前介绍的基本 HTTP 操作，Go 语言标准库也暴露了比较低层的 HTTP 相关库，让开发者可以基于这些库灵活定制 HTTP 服务器和使用 HTTP 服务。

1）自定义 http.Client

前面使用的 http.Get()、http.Post()、http.PostForm()和 http.Head()方法其实都是在 http.DefaultClient 的基础上进行调用的，比如 http.Get()等价于 http.Default-Client.Get()，依此类推。

http.DefaultClient 在字面上就向我们传达了一个信息，既然存在默认的 Client，那么 HTTP Client 大概是可以自定义的。实际上确实如此，在 net/http 包中，的确提供了 Client 类型。下面来看一看 http.Client 类型的结构：

```
type Client struct {
    // Transport 用于确定 HTTP 请求的创建机制
    // 如果为空，将会使用 DefaultTransport
    Transport RoundTripper
    // CheckRedirect 定义重定向策略
    // 如果 CheckRedirect 不为空，客户端将在跟踪 HTTP 重定向前调用该函数
    // 两个参数 req 和 via 分别为即将发起的请求和已经发起的所有请求，最早的已发起请求在最前面
```

```
    // 如果 CheckRedirect 返回错误，客户端将直接返回错误，不会再发起该请求
    // 如果 CheckRedirect 为空，Client 将采用一种确认策略，将在 10 个连续请求后终止
    CheckRedirect func(req *Request, via []*Request) error
    // 如果 Jar 为空，Cookie 将不会在请求中发送，并会在响应中被忽略
    Jar CookieJar
}
```

在 Go 语言标准库中，http.Client 类型包含了 3 个公开数据成员：Transport RoundTripper、CheckRedirect func(req *Request, via []*Request) error、Jar CookieJar

其中 Transport 类型必须实现 http.RoundTripper 接口。Transport 指定了执行一个 HTTP 请求的运行机制，倘若不指定具体的 Transport，默认会使用 http.DefaultTransport，这意味着 http.Transport 也是可以自定义的。net/http 包中的 http.Transport 类型实现了 http.RoundTripper 接口。

CheckRedirect 函数指定处理重定向的策略。当使用 HTTP Client 的 Get()或者 Head()方法发送 HTTP 请求时，若响应返回的状态码为 30x（比如 301/302/303/307），HTTP Client 会在遵循跳转规则之前先调用这个 CheckRedirect 函数。

Jar 可用于在 HTTP Client 中设定 Cookie，Jar 的类型必须实现了 http.CookieJar 接口，该接口预定义了 SetCookies()和 Cookies()两个方法。如果 HTTP Client 中没有设定 Jar，Cookie 将被忽略而不会发送到客户端。实际上一般都用 http.SetCookie()方法来设定 Cookie。

使用自定义的 http.Client 及其 Do()方法，可以非常灵活地控制 HTTP 请求，比如发送自定义 HTTP Header 或改写重定向策略等。创建自定义的 HTTP Client 非常简单，具体代码如下：

```
client := &http.Client {
      CheckRedirect: redirectPolicyFunc,
}
resp, err := client.Get("http://example.com")
// ...
req, err := http.NewRequest("GET", "http://example.com", nil)
// ...
req.Header.Add("User-Agent", "Our Custom User-Agent")
req.Header.Add("If-None-Match", `W/"TheFileEtag"`)
resp, err := client.Do(req)
// ...
```

2）自定义 http.Transport

在 http.Client 类型的结构定义中，看到的第一个数据成员就是一个 http.Transport 对象，

该对象指定执行一个 HTTP 请求时的运行规则。下面来看看 http.Transport 类型的具体结构：

```
type Transport struct {
    // Proxy 指定用于针对特定请求返回代理的函数
    // 如果该函数返回一个非空的错误，请求将终止并返回该错误
    // 如果 Proxy 为空或者返回一个空的 URL 指针，将不使用代理
    Proxy func(*Request) (*url.URL, error)
    // Dial 指定用于创建 TCP 连接的 dail()函数
    // 如果 Dial 为空，将默认使用 net.Dial()函数
    Dial func(net, addr string) (c net.Conn, err error)
    // TLSClientConfig 指定用于 tls.Client 的 TLS 配置
    // 如果为空则使用默认配置
    TLSClientConfig    *tls.Config
    DisableKeepAlives  bool
    DisableCompression bool
    // 如果 MaxIdleConnsPerHost 为非零值，它用于控制每个 host 所需要保持的最大空闲连接数
    // 如果该值为空，则使用 DefaultMaxIdleConnsPerHost
    MaxIdleConnsPerHost int
    // ...
}
```

在上面的代码中，定义了 http.Transport 类型中的公开数据成员，下面详细说明其中的各行代码：

```
Proxy func(*Request) (*url.URL, error)
```

Proxy 指定了一个代理方法，该方法接收一个*Request 类型的请求实例作为参数并返回一个最终的 HTTP 代理。如果 Proxy 未指定或者返回的*URL 为零值，将不会有代理被启用：

```
Dial func(net, addr string) (c net.Conn, err error)
```

Dial 指定具体的 dial()方法来创建 TCP 连接。如果不指定，默认将使用 net.Dial()方法：

```
TLSClientConfig *tls.Config
```

SSL 连接专用，TLSClientConfig 指定 tls.Client 所用的 TLS 配置信息，如果不指定，也会使用默认的配置：

```
DisableKeepAlives bool
```

是否取消长连接，默认值为 false，即启用长连接：

```
DisableCompression bool
```

是否取消压缩（GZip），默认值为 false，即启用压缩：

```
MaxIdleConnsPerHost int
```

指定与每个请求的目标主机之间的最大非活跃连接（keep-alive）数量。如果不指定，默认使用 DefaultMaxIdleConnsPerHost 的常量值。

除了 http.Transport 类型中定义的公开数据成员以外，它同时还提供了几个公开的成员方法：

```
// 该方法用于关闭所有非活跃的连接
func(t *Transport) CloseIdleConnections()
// 该方法可用于注册并启用一个新的传输协议
// 比如 WebSocket 的传输协议标准（ws），或者 FTP、File 协议等
func(t *Transport) RegisterProtocol(scheme string, rt RoundTripper)
// 用于实现 http.RoundTripper 接口
func(t *Transport) RoundTrip(req *Request) (resp *Response, err error)
```

自定义 http.Transport 也很简单，如下列代码所示：

```
tr := &http.Transport{
    TLSClientConfig: &tls.Config{RootCAs: pool},
    DisableCompression: true,
}
client := &http.Client{Transport: tr}
resp, err := client.Get("https://example.com")
```

Client 和 Transport 在执行多个 goroutine 的并发过程中都是安全的，但出于性能考虑，应当创建一次后反复使用。

11.2.2 HTTP 服务器端

本节将介绍 HTTP 服务器端技术，包括如何处理 HTTP 请求和 HTTPS 请求。

1. 处理 HTTP 请求

使用 net/http 包提供的 http.ListenAndServe()方法，可以在指定的地址进行监听，开启一个 HTTP，在服务器端该方法的原型如下：

```
func ListenAndServe(addr string, handler Handler) error
```

该方法用于在指定的 TCP 网络地址 addr 进行监听，然后调用服务器端处理程序来处理传入的连接请求。该方法有两个参数：第一个参数 addr 即监听地址；第二个参数表示服务器端处理程序，通常为空，这意味着服务器端调用 http.DefaultServeMux 进行处理，而服务器端编写的业务逻辑处理程序 http.Handle()或 http.HandleFunc()默认注入 http.DefaultServeMux 中，具体代码如下：

```
http.Handle("/foo", fooHandler)
http.HandleFunc("/bar", func(w http.ResponseWriter, r *http.Request) {
      fmt.Fprintf(w, "Hello, %q", html.EscapeString(r.URL.Path))
})
log.Fatal(http.ListenAndServe(":8080", nil))
```

如果想更多地控制服务器端的行为，可以自定义 http.Server，代码如下：

```
s := &http.Server{
      Addr:":8080",
      Handler:myHandler,
      ReadTimeout:10 * time.Second,
      WriteTimeout:10 * time.Second,
      MaxHeaderBytes: 1 << 20,
}
log.Fatal(s.ListenAndServe())
```

2. 处理 HTTPS 请求

net/http 包还提供 http.ListenAndServeTLS()方法，用于处理 HTTPS 连接请求：

```
func ListenAndServeTLS(addr string, certFile string, keyFile string, handler
Handler) error
```

ListenAndServeTLS()和 ListenAndServe()的行为一致，区别在于只处理 HTTPS 请求。此外，服务器上必须存在包含证书和与之匹配的私钥的相关文件，比如 certFile 对应 SSL 证书文件存放路径，keyFile 对应证书私钥文件路径。如果证书是由证书颁发机构签署的，certFile 参数指定的路径必须是存放在服务器上的经由 CA 认证过的 SSL 证书。

开启 SSL 监听服务也很简单，如下列代码所示：

```
http.Handle("/foo", fooHandler)
http.HandleFunc("/bar", func(w http.ResponseWriter, r *http.Request) {
      fmt.Fprintf(w, "Hello, %q", html.EscapeString(r.URL.Path))
})
log.Fatal(http.ListenAndServeTLS(":10443", "cert.pem", "key.pem", nil))
```

或者是：

```
ss := &http.Server{
    Addr: ":10443",
    Handler: myHandler,
    ReadTimeout: 10 * time.Second,
    WriteTimeout: 10 * time.Second,
    MaxHeaderBytes: 1 << 20,
}
log.Fatal(ss.ListenAndServeTLS("cert.pem", "key.pem"))
```

11.3 RPC 编程

前面几节介绍了如何基于 Socket 和 HTTP 来编写网络应用，通过学习了解了 Socket 和 HTTP 采用的是类似“信息交换”模式，即客户端发送一条信息到服务器端。一般来说服务器端都会返回一定的信息以表示响应。客户端和服务器端之间约定了交互信息的格式，以便双方都能解析交互所产生的信息。但是很多独立的应用并没有采用这种模式，而是采用类似常规的函数调用的方式来完成想要的功能。

RPC 就是想实现函数调用模式的网络化。客户端就像调用本地函数一样，然后客户端把这些参数打包之后通过网络传递到服务器端，服务器端解包到处理过程中执行，然后执行的结果反馈给客户端。

RPC（远程过程调用协议），是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。它假定某些传输协议的存在，如 TCP 或 UDP，以便为通信程序之间携带信息数据，通过它可以使函数调用模式网络化。在 OSI 网络通信模型中，RPC 跨越了传输层和应用层。RPC 使得开发包括网络分布式多程序在内的应用程序更加容易。

运行时，一次客户机对服务器的 RPC 调用，其内部操作大致有如下 10 步。

（1）调用客户端句柄，执行传送参数。

（2）调用本地系统内核发送网络消息。

（3）消息传送到远程主机。

（4）服务器句柄得到消息并取得参数。

（5）执行远程过程。

（6）执行的过程将结果返回服务器句柄。

（7）服务器句柄返回结果，调用远程系统内核。

（8）消息传回本地主机。

（9）客户句柄由内核接收消息。

（10）客户接收句柄返回的数据。

11.3.1　Go RPC

Go 语言标准包中已经提供了对 RPC 的支持，而且支持三个级别的 RPC：TCP、HTTP、JSONRPC。但 Go 语言的 RPC 包是独一无二的 RPC，它和传统的 RPC 系统不同，它只支持 Go 语言开发的服务器与客户端之间的交互，因为在内部，它们采用了 Gob 来编码。

Go RPC 的函数只有符合下面的条件才能被远程访问，不然会被忽略，详细的要求如下：

- 函数必须是导出的（首字母大写）。
- 必须有两个导出类型的参数。
- 第一个参数是接收的参数，第二个参数是返回给客户端的参数，第二个参数必须是指针类型。
- 函数还要有一个返回值 error。

举个例子，正确的 RPC 函数格式如下：

```
func (t *T) MethodName(argType T1, replyType *T2) error
```

T、T1 和 T2 类型必须能被 encoding/gob 包编解码。任何的 RPC 都需要通过网络来传递数据，Go RPC 可以利用 HTTP 和 TCP 来传递数据，利用 HTTP 的好处是可以直接复用 net/http 里面的一些函数。详细的例子请看下面的实现。

11.3.2　HTTP RPC

HTTP 的服务器端代码实现如下：

```
// 示例代码 11-9
package main

import (
      "errors"
      "fmt"
      "net/http"
      "net/rpc"
)

type Args struct {
      A, B int
}

type Quotient struct {
      Quo, Rem int
}
```

```
type Arith int

func (t *Arith) Multiply(args *Args, reply *int) error {
      *reply = args.A * args.B
      return nil
}

func (t *Arith) Divide(args *Args, quo *Quotient) error {
      if args.B == 0 {
            return errors.New("除数不能为零")
      }
      quo.Quo = args.A / args.B
      quo.Rem = args.A % args.B
      return nil
}

func main() {
      arith := new(Arith)
      rpc.Register(arith)
      rpc.HandleHTTP()

      err := http.ListenAndServe(":1234", nil)
      if err != nil {
            fmt.Println(err.Error())
      }
}
```

通过上面的例子可以看到，注册了一个 Arith 的 RPC 服务，然后通过 rpc.HandleHTTP 函数把该服务注册到 HTTP 协议上，最后就可以利用 HTTP 的方式来传递数据。

请看下面的客户端代码：

```
// 示例代码 11-10
package main

import (
      "fmt"
      "log"
      "net/rpc"
      "os"
)

type Args struct {
      A, B int
}

type Quotient struct {
```

```
        Quo, Rem int
    }

    func main() {
        if len(os.Args) != 2 {
            fmt.Println("用法: ", os.Args[0], "server")
            os.Exit(1)
        }
        serverAddress := os.Args[1]

        client, err := rpc.DialHTTP("tcp", serverAddress+":1234")
        if err != nil {
            log.Fatal("Dial 错误: ", err)
        }
        // Synchronous call
        args := Args{17, 8}
        var reply int
        err = client.Call("Arith.Multiply", args, &reply)
        if err != nil {
            log.Fatal("arith 错误: ", err)
        }
        fmt.Printf("Arith: %d*%d=%d\n", args.A, args.B, reply)

        var quot Quotient
        err = client.Call("Arith.Divide", args, &quot)
        if err != nil {
            log.Fatal("arith 错误: ", err)
        }
        fmt.Printf("Arith: %d/%d=%d 余 %d\n", args.A, args.B, quot.Quo, quot.Rem)

    }
```

把上面的服务器端和客户端的代码分别编译，先把服务器端开启，再开启客户端，输入代码，就会输出如下信息：

```
$ ./http_c localhost
Arith: 17*8=136
Arith: 17/8=2 余 1
```

通过上面的调用可以看到参数和返回值是我们定义的 struct 类型，在服务器端把它们当作调用函数的参数的类型，在客户端作为 client.Call 的第 2 和第 3 两个参数的类型。客户端最重要的就是这个 Call 函数，它有 3 个参数，第 1 个是要调用的函数的名字，第 2 个是要传递的参数，第 3 个要返回的参数（注意是指针类型）。通过上面的代码例子可以发现，使用 Go 语言的 RPC 实现相当简单方便。

11.3.3 TCP RPC

上面实现了基于 HTTP 协议的 RPC，接下来要实现基于 TCP 协议的 RPC，服务器端的实现代码如下所示：

```
// 示例代码 11-11
package main

import (
	"errors"
	"fmt"
	"net"
	"net/rpc"
	"os"
)

type Args struct {
	A, B int
}

type Quotient struct {
	Quo, Rem int
}

type Arith int

func (t *Arith) Multiply(args *Args, reply *int) error {
	*reply = args.A * args.B
	return nil
}

func (t *Arith) Divide(args *Args, quo *Quotient) error {
	if args.B == 0 {
		return errors.New("除数不能为零")
	}
	quo.Quo = args.A / args.B
	quo.Rem = args.A % args.B
	return nil
}

func main() {

	arith := new(Arith)
	rpc.Register(arith)

	tcpAddr, err := net.ResolveTCPAddr("tcp", ":1234")
	checkError(err)
```

```
        listener, err := net.ListenTCP("tcp", tcpAddr)
        checkError(err)

        for {
                conn, err := listener.Accept()
                if err != nil {
                        continue
                }
                rpc.ServeConn(conn)
        }

}

func checkError(err error) {
        if err != nil {
                fmt.Println("错误: ", err.Error())
                os.Exit(1)
        }
}
```

上面这个代码和 HTTP 的服务器相比，不同之处在于：在此处采用了 TCP 协议，然后需要自己控制连接，当有客户端连接上来后，需要把这个连接交给 RPC 来处理。

如果留心了，会发现这是一个阻塞型的单用户的程序，如果想要实现多并发，那么可以使用 goroutine 来实现，前面在介绍 Socket 一节中已经介绍过如何处理 goroutine。下面展现了 TCP 实现的 RPC 客户端：

```
// 示例代码 11-12
package main

import (
        "fmt"
        "log"
        "net/rpc"
        "os"
)

type Args struct {
        A, B int
}

type Quotient struct {
        Quo, Rem int
}

func main() {
```

```
    if len(os.Args) != 2 {
        fmt.Println("用法: ", os.Args[0], "server:port")
        os.Exit(1)
    }
    service := os.Args[1]

    client, err := rpc.Dial("tcp", service)
    if err != nil {
        log.Fatal("Dial 错误: ", err)
    }
    // Synchronous call
    args := Args{17, 8}
    var reply int
    err = client.Call("Arith.Multiply", args, &reply)
    if err != nil {
        log.Fatal("arith 错误: ", err)
    }
    fmt.Printf("Arith: %d*%d=%d\n", args.A, args.B, reply)

    var quot Quotient
    err = client.Call("Arith.Divide", args, &quot)
    if err != nil {
        log.Fatal("arith 错误: ", err)
    }
    fmt.Printf("Arith: %d/%d=%d 余 %d\n", args.A, args.B, quot.Quo, quot.Rem)
}
```

这个客户端代码和 HTTP 的客户端代码对比，唯一的区别在于一个是 DialHTTP，一个是 Dial(tcp)，其他处理一模一样。

11.3.4 JSON RPC

JSON RPC 是数据编码采用了 JSON，而不是 Gob 编码，其他和上面介绍的 RPC 概念一模一样，下面来演示如何使用 Go 语言提供的 json-rpc 标准包。服务器端代码的实现如下：

```
// 示例代码 11-13
package main

import (
    "errors"
    "fmt"
    "net"
    "net/rpc"
    "net/rpc/jsonrpc"
    "os"
```

```
)

type Args struct {
      A, B int
}

type Quotient struct {
      Quo, Rem int
}

type Arith int

func (t *Arith) Multiply(args *Args, reply *int) error {
      *reply = args.A * args.B
      return nil
}

func (t *Arith) Divide(args *Args, quo *Quotient) error {
      if args.B == 0 {
            return errors.New("除数不能为零")
      }
      quo.Quo = args.A / args.B
      quo.Rem = args.A % args.B
      return nil
}

func main() {

      arith := new(Arith)
      rpc.Register(arith)

      tcpAddr, err := net.ResolveTCPAddr("tcp", ":1234")
      checkError(err)

      listener, err := net.ListenTCP("tcp", tcpAddr)
      checkError(err)

      for {
            conn, err := listener.Accept()
            if err != nil {
                  continue
            }
            jsonrpc.ServeConn(conn)
      }

}

func checkError(err error) {
```

```
        if err != nil {
                fmt.Println("错误：", err.Error())
                os.Exit(1)
        }
}
```

通过示例可以看出 json-rpc 是基于 TCP 协议实现的，目前它还不支持 HTTP 方式。

客户端的实现代码如下：

```
// 示例代码 11-14
package main

import (
        "fmt"
        "log"
        "net/rpc/jsonrpc"
        "os"
)

type Args struct {
        A, B int
}

type Quotient struct {
        Quo, Rem int
}

func main() {
        if len(os.Args) != 2 {
                fmt.Println("用法：", os.Args[0], "server:port")
                log.Fatal(1)
        }
        service := os.Args[1]

        client, err := jsonrpc.Dial("tcp", service)
        if err != nil {
                log.Fatal("Dial 错误：", err)
        }
        // 同步调用
        args := Args{17, 8}
        var reply int
        err = client.Call("Arith.Multiply", args, &reply)
        if err != nil {
                log.Fatal("arith 错误：", err)
        }
        fmt.Printf("Arith: %d*%d=%d\n", args.A, args.B, reply)

        var quot Quotient
```

```
        err = client.Call("Arith.Divide", args, &quot)
        if err != nil {
              log.Fatal("arith 错误: ", err)
        }
        fmt.Printf("Arith: %d/%d=%d 余 %d\n", args.A, args.B, quot.Quo, quot.Rem)

}
```

11.3.5　RPC 接口

Go 语言的 net/rpc 很灵活，它在数据传输前后实现了编码解码器的接口定义，这意味着开发者可以自定义数据的传输方式以及 RPC 服务器端与客户端之间的交互行为。

RPC 提供的编码解码器接口如下：

```
type ClientCodec interface {
    WriteRequest(*Request, interface{}) error
    ReadResponseHeader(*Response) error
    ReadResponseBody(interface{}) error
    Close() error
}

type ServerCodec interface {
    ReadRequestHeader(*Request) error
    ReadRequestBody(interface{}) error
    WriteResponse(*Response, interface{}) error
    Close() error
}
```

接口 ClientCodec 定义了 RPC 客户端如何在一个 RPC 会话中发送请求和读取响应。客户端程序通过 WriteRequest()方法将一个请求写入 RPC 连接中，并通过 ReadResponseHeader()和 ReadResponseBody()读取服务器端的响应信息。当整个过程执行完毕后，再通过 Close()方法关闭该连接。

接口 ServerCodec 定义了 RPC 服务器端如何在一个 RPC 会话中接收请求并发送响应。服务器端程序通过 ReadRequestHeader()和 ReadRequestBody()方法从一个 RPC 连接中读取请求信息，然后再通过 WriteResponse()方法向该连接中的 RPC 客户端发送响应。在完成该过程后，通过 Close()方法来关闭连接。

通过实现上述接口，可以自定义数据传输前后的编码解码方式，而不仅仅局限于 Gob。同样，可以自定义 RPC 服务器端和客户端的交互行为。实际上，Go 语言标准库提供的 net、rpc、json 包，就是一套实现了 rpc.ClientCodec 和 rpc.ServerCodec 接口的 JSON-RPC 模块。

11.4 数据库

对许多 Web 应用程序而言，数据库都是其核心所在。数据库几乎可以用来存储想查询和修改的任何信息，比如用户信息、产品目录或者新闻列表等。

Go 语言没有内置的驱动支持任何的数据库，但是 Go 语言定义了 database/sql 接口，用户可以基于驱动接口开发相应数据库的驱动，后面会介绍 Go 语言设计的一些驱动、Go 语言是如何设计数据库驱动接口的。本节会介绍目前用得较多的一些关系型数据驱动以及如何使用它们。

目前 NoSQL 已经成为 Web 开发的一个潮流，很多应用采用了 NoSQL 作为数据库，而不是以前的缓存，最后还会介绍 MongoDB 和 Redis 两种 NoSQL 数据库。

如下网址提供了惯用的范例及详细的说明：http://go- database-sql.org。

11.4.1 database/sql 接口

Go 语言与 PHP 不同的地方是 Go 语言官方没有提供数据库驱动，而是为开发数据库驱动定义了一些标准接口，开发者可以根据定义的接口来开发相应的数据库驱动，这样做有一个好处，只要是按照标准接口开发的代码，以后需要迁移数据库时，不需要做任何修改。

下面会逐一分析 Go 语言定义的标准接口。

1. sql.Register

这个存在于 database/sql 中的函数是用来注册数据库驱动的，当第三方开发者开发数据库驱动时，都会实现 init 函数，在 init 里面会调用这个 Register(name string, driver driver.Driver)完成本驱动的注册。

下面来看看 mymysql、sqlite3 的驱动里面都是怎样调用的：

```
// https://github.com/mattn/go-sqlite3
func init() {
      sql.Register("sqlite3", &SQLiteDriver{})
}

// https://github.com/mikespook/mymysql
// Driver automatically registered in database/sql
var d = Driver{proto: "tcp", raddr: "127.0.0.1:3306"}
func init() {
```

```
        Register("SET NAMES utf8")
        sql.Register("mymysql", &d)
}
```

可以看到第三方数据库驱动都是通过调用这个函数来注册自己的数据库驱动名称以及相应的 driver 实现的。在 database/sql 内部通过一个 map 来存储用户定义的相应驱动。

```
var drivers = make(map[string]driver.Driver)
drivers[name] = driver
```

因此通过 database/sql 的注册函数可以同时注册多个数据库驱动，只要不重复。

在使用 database/sql 接口和第三方库的时候经常看到如下代码：

```
import (
    "database/sql"
    _ "github.com/mattn/go-sqlite3"
)
```

新手都会被这个_所迷惑，其实这个就是 Go 语言设计的巧妙之处。在变量赋值的时候经常看到这个符号，它是用来忽略变量赋值的占位符的，那么包引入用到这个符号也是相似的作用，这里使用_的意思是引入后面的包名而不直接使用这个包中定义的函数、变量等资源。

在前面函数章节中介绍过 init 函数的初始化过程，包在引入的时候会自动调用包的 init 函数以完成对包的初始化。因此，引入上面的数据库驱动包之后会自动去调用 init 函数，然后在 init 函数里面注册这个数据库驱动，这样就可以在接下来的代码中直接使用这个数据库驱动了。

2. driver.Driver

Driver 是一个数据库驱动的接口，它定义了一个方法 Open(name string)，这个方法返回一个数据库的 Conn 接口：

```
type Driver interface {
        Open(name string) (Conn, error)
}
```

返回的 Conn 只能用来进行一次 goroutine 的操作，也就是说不能把这个 Conn 应用于 Go 语言的多个 goroutine 里面。如下代码会出现错误：

```
...
go goroutineA (Conn)  //执行查询操作
go goroutineB (Conn)  //执行插入操作
...
```

上面这样的代码可能会使 Go 语言不知道某个操作究竟是由哪个 goroutine 发起的，从而导致数据混乱，比如可能会把 goroutineA 里面执行的查询操作的结果返回给 goroutineB，从而使 goroutineB 错误地把此结果当成自己执行的插入数据。

第三方驱动都会定义这个函数，它会解析 name 参数来获取相关数据库的连接信息，解析完成后，它将使用此信息来初始化一个 Conn 并返回它。

3. driver.Conn

Conn 是一个数据库连接的接口定义，它定义了一系列方法，这个 Conn 只能应用在一个 goroutine 里面，不能使用在多个 goroutine 里面，详情请参考上面的说明。

```
type Conn interface {
      Prepare(query string) (Stmt, error)
      Close() error
      Begin() (Tx, error)
}
```

Prepare 函数返回与当前连接相关的执行 SQL 语句的准备状态，可以进行查询、删除等操作。

Close 函数关闭当前的连接，执行释放连接拥有的资源等清理工作。因为驱动实现了 database/sql 里面建议的 conn pool，所以不用再去实现缓存 conn 之类的，这样容易引起问题。

Begin 函数返回一个代表事务处理的 Tx，通过它可以进行查询、更新等操作，或者对事务进行回滚、递交。

4. driver.Stmt

Stmt 是一种准备好的状态，和 Conn 相关联，而且只能应用于一个 goroutine 中，不能应用于多个 goroutine：

```
type Stmt interface {
      Close() error
      NumInput() int
      Exec(args []Value) (Result, error)
      Query(args []Value) (Rows, error)
}
```

Close 函数关闭当前的连接状态，但是如果当前正在执行 query，query 还是有效返回 rows 数据。

NumInput 函数返回当前预留参数的个数，当返回>=0 时数据库驱动就会智能检查调

用者的参数。当数据库驱动包不知道预留参数的时候，返回-1。

Exec 函数执行 Prepare 准备好的 SQL，传入参数执行 Update/Insert 等操作，返回 Result 数据；Query 函数执行 Prepare 准备好的 SQL，传入需要的参数执行 select 操作，返回 Rows 结果集。

5. driver.Tx

事务处理一般就两个过程，递交或者回滚，数据库驱动里面也只需要实现这两个函数：

```
type Tx interface {
      Commit() error
      Rollback() error
}
```

这两个函数一个用来递交事务，一个用来回滚事务。

6. driver.Execer

这是一个 Conn 可选择实现的接口：

```
type Execer interface {
      Exec(query string, args []Value) (Result, error)
}
```

如果这个接口没有定义，那么调用 DB.Exec，就会首先调用 Prepare 返回 Stmt，然后执行 Stmt 的 Exec，最后关闭 Stmt。

7. driver.Result

这个是执行 Update/Insert 等操作返回的结果接口定义：

```
type Result interface {
      LastInsertId() (int64, error)
      RowsAffected() (int64, error)
}
```

LastInsertId 函数返回由数据库执行插入操作得到的自增 ID 号。RowsAffected 函数返回 query 操作影响的数据条目数。

8. driver.Rows

Rows 是执行查询返回的结果集接口定义：

```
type Rows interface {
      Columns() []string
      Close() error
```

```
        Next(dest []Value) error
}
```

Columns 函数返回查询数据库表的字段信息，这个返回的 slice 和 SQL 查询的字段一一对应，而不是返回整个表的所有字段。Close 函数用来关闭 Rows 迭代器。

Next 函数用来返回下一条数据，把数据赋值给 dest。dest 里面的元素必须是 driver.Value 的值（除了 string），返回的数据里面所有的 string 都必须转换成[]byte。如果最后没数据了，Next 函数返回 io.EOF。

9. driver.RowsAffected

RowsAffected 其实就是一个 int64 的别名，但是它实现了 Result 接口，用来底层实现 Result 的表示方式：

```
type RowsAffected int64
func (RowsAffected) LastInsertId() (int64, error)
func (v RowsAffected) RowsAffected() (int64, error)
```

10. driver.Value

Value 其实就是一个空接口，它可以容纳任何的数据：

```
type Value interface{}
```

drive 的 Value 是驱动必须能够操作的 Value，Value 要么是 nil，要么是下面的任意一种：

```
int64
float64
bool
[]byte
string   [*] // 除了 Rows.Next，返回的不能是 string
time.Time
```

11. driver.ValueConverter

ValueConverter 接口定义了如何把一个普通的值转化成 driver.Value 的接口：

```
type ValueConverter interface {
      ConvertValue(v interface{}) (Value, error)
}
```

在开发的数据库驱动包里面实现这个接口的函数在很多地方会使用到，这个 ValueConverter 有很多好处：

- 转化 driver.value 到数据库表相应的字段，例如 int64 的数据如何转化成数据库表 uint16 字段。

- 把数据库查询结果转化成 driver.Value 值。
- 在 scan 函数里面如何把 driver.Value 值转化成用户定义的值。

12. driver.Valuer

Valuer 接口定义了返回一个 driver.Value 的方式：

```
type Valuer interface {
      Value() (Value, error)
}
```

很多类型都实现了这个 Value 方法，用于自身与 driver.Value 的转化。

通过上面的讲解，读者应该对于驱动的开发有了一个基本的了解了，一个驱动只要实现了这些接口就能完成增删查改等基本操作，剩下的就是与相应的数据库进行数据交互等细节问题了，在此不再赘述。

11.4.2　使用 MySQL 数据库

目前 Internet 上流行的网站构架方式是 LAMP/LNMP，其中的 M 即 MySQL，作为数据库，MySQL 以免费、开源、使用方便为优势成为了很多 Web 开发的后端数据库存储引擎。

目前 Go 语言中支持 MySQL 的驱动比较多，有些是支持 database/sql 标准，而有些是采用了自己的实现接口，常用的有如下几种：

- https://github.com/go-sql-driver/mysql 支持 database/sql，全部采用 Go 语言编写。
- https://github.com/ziutek/mymysql 支持 database/sql，也支持自定义的接口，全部采用 Go 语言编写。
- https://github.com/Philio/GoMySQL 不支持 database/sql，自定义接口，全部采用 Go 语言编写。

接下来的例子主要以第一个驱动为例，主要理由如下：

- 这个驱动比较新，维护得比较好。
- 完全支持 database/sql 接口。
- 支持 keepalive，保持长连接。

接下来几节的内容都将采用同一个数据库表结构：数据库 test，用户表 userinfo，关联用户信息表 userdetail。

仅供非商业用途或交流学习使用


```
CREATE TABLE `userinfo` (
      `uid` INT(10) NOT NULL AUTO_INCREMENT,
      `username` VARCHAR(64) NULL DEFAULT NULL,
      `departname` VARCHAR(64) NULL DEFAULT NULL,
      `created` DATE NULL DEFAULT NULL,
      PRIMARY KEY (`uid`)
);

CREATE TABLE `userdetail` (
      `uid` INT(10) NOT NULL DEFAULT '0',
      `intro` TEXT NULL,
      `profile` TEXT NULL,
      PRIMARY KEY (`uid`)
)
```

如下示例将示范如何使用 database/sql 接口对数据库表进行增删改查操作：

```
// 示例代码 11-15
package main

import (
      "database/sql"
      "fmt"
      //"time"

      _ "github.com/go-sql-driver/mysql"
)

func main() {
      db, err := sql.Open("mysql", "zuolan:zuolan@/test?charset=utf8")
      checkErr(err)

      //插入数据
      stmt, err := db.Prepare("INSERT userinfo SET username=?,departname=?,created=?")
      checkErr(err)

      res, err := stmt.Exec("张三", "研发部门", "2017-09-09")
      checkErr(err)

      id, err := res.LastInsertId()
      checkErr(err)

      fmt.Println(id)
      //更新数据
      stmt, err = db.Prepare("update userinfo set username=? where uid=?")
      checkErr(err)

      res, err = stmt.Exec("zuolanupdate", id)
      checkErr(err)
```

仅供非商业用途或交流学习使用


```
	affect, err := res.RowsAffected()
	checkErr(err)

	fmt.Println(affect)

	//查询数据
	rows, err := db.Query("SELECT * FROM userinfo")
	checkErr(err)

	for rows.Next() {
		var uid int
		var username string
		var department string
		var created string
		err = rows.Scan(&uid, &username, &department, &created)
		checkErr(err)
		fmt.Println(uid)
		fmt.Println(username)
		fmt.Println(department)
		fmt.Println(created)
	}

	//删除数据
	stmt, err = db.Prepare("delete from userinfo where uid=?")
	checkErr(err)

	res, err = stmt.Exec(id)
	checkErr(err)

	affect, err = res.RowsAffected()
	checkErr(err)

	fmt.Println(affect)

	db.Close()

}

func checkErr(err error) {
	if err != nil {
		panic(err)
	}
}
```

通过上面的代码可以看出，Go 语言操作 MySQL 数据库是很方便的。

sql.Open()函数用来打开一个注册过的数据库驱动，go-sql-driver 中注册了 MySQL 这

仅供非商业用途或交流学习使用

个数据库驱动，第二个参数是 DSN（Data Source Name），它是 go-sql-driver 定义的一些数据库连接和配置信息。它支持如下格式：

```
user@unix(/path/to/socket)/dbname?charset=utf8
user:password@tcp(localhost:5555)/dbname?charset=utf8
user:password@/dbname
user:password@tcp([de:ad:be:ef::ca:fe]:80)/dbname
```

- db.Prepare()函数用来返回准备要执行的 SQL 操作，然后返回准备完毕的执行状态。
- db.Query()函数用来直接执行 SQL 返回 Rows 结果。
- stmt.Exec()函数用来执行 stmt 准备好的 SQL 语句。

可以看到，传入的参数都是=?对应的数据，这样做的方式可以在一定程度上防止 SQL 注入。

11.4.3 使用 SQLite 数据库

SQLite 是一个开源的嵌入式关系型数据库，实现自包容、零配置、支持事务的 SQL 数据库引擎。其特点是高度便携、使用方便、结构紧凑、高效、可靠。与其他数据库管理系统不同，SQLite 的安装和运行非常简单，在大多数情况下，只要确保 SQLite 的二进制文件存在即可开始创建、连接和使用数据库。

如果正在寻找一个嵌入式数据库项目或解决方案，SQLite 绝对值得考虑。SQLite 可以说是开源的 Access。

Go 语言支持 SQLite 的驱动也比较多，但是很多都不支持 database/sql 接口：

- https://github.com/mattn/go-sqlite3 支持 database/sql 接口，基于 cgo（关于 cgo 的知识请参看官方文档）。
- https://github.com/feyeleanor/gosqlite3 不支持 database/sql 接口，基于 cgo。
- https://github.com/phf/go-sqlite3 不支持 database/sql 接口，基于 cgo。

目前支持 database/sql 的 SQLite 数据库驱动比较少，采用标准接口有利于以后出现更好的驱动的时候做迁移，因此本节后面都采用第一个驱动。

示例的数据库表结构相应的建表 SQL 如下：

```
CREATE TABLE `userinfo` (
      `uid` INTEGER PRIMARY KEY AUTOINCREMENT,
      `username` VARCHAR(64) NULL,
      `departname` VARCHAR(64) NULL,
```

仅供非商业用途或交流学习使用


```
    `created` DATE NULL
);

CREATE TABLE `userdeatail` (
    `uid` INT(10) NULL,
    `intro` TEXT NULL,
    `profile` TEXT NULL,
    PRIMARY KEY (`uid`)
);
```

看下面 Go 语言程序是如何操作数据库表进行数据增删改查的：

```
// 示例代码 11-16
package main

import (
    "database/sql"
    "fmt"
    "time"

    _ "github.com/mattn/go-sqlite3"
)

func main() {
    db, err := sql.Open("sqlite3", "./demo.db")
    checkErr(err)

    //插入数据
    stmt, err := db.Prepare("INSERT INTO userinfo(username, departname, created)
values(?,?,?)")
    checkErr(err)

    res, err := stmt.Exec("张三", "研发部门", "2017-09-09")
    checkErr(err)

    id, err := res.LastInsertId()
    checkErr(err)

    fmt.Println(id)
    //更新数据
    stmt, err = db.Prepare("update userinfo set username=? where uid=?")
    checkErr(err)

    res, err = stmt.Exec("zuolanupdate", id)
    checkErr(err)

    affect, err := res.RowsAffected()
    checkErr(err)
```

仅供非商业用途或交流学习使用


```
        fmt.Println(affect)

        //查询数据
        rows, err := db.Query("SELECT * FROM userinfo")
        checkErr(err)

        for rows.Next() {
                var uid int
                var username string
                var department string
                var created time.Time
                err = rows.Scan(&uid, &username, &department, &created)
                checkErr(err)
                fmt.Println(uid)
                fmt.Println(username)
                fmt.Println(department)
                fmt.Println(created)
        }

        //删除数据
        stmt, err = db.Prepare("delete from userinfo where uid=?")
        checkErr(err)

        res, err = stmt.Exec(id)
        checkErr(err)

        affect, err = res.RowsAffected()
        checkErr(err)

        fmt.Println(affect)

        db.Close()

}

func checkErr(err error) {
        if err != nil {
                panic(err)
        }
}
```

可以看到上面的代码和 MySQL 例子里面的代码几乎一模一样，唯一改变的就是导入的驱动改变了，然后调用 sql.Open 时采用了 SQLite 的方式打开。

11.4.4 使用 PostgreSQL 数据库

PostgreSQL 是一个自由的对象—关系型数据库服务器（数据库管理系统），它在灵活

仅供非商业用途或交流学习使用

的 BSD 风格许可证下发行。它提供了相对其他开源数据库系统（比如 MySQL 和 Firebird），以及对专有系统（比如 Oracle、Sybase、IBM 的 DB2 和 Microsoft SQL Server）的一种选择。

PostgreSQL 和 MySQL 比较，它更加庞大一点，因为它是用来替代 Oracle 而设计的，所以在企业应用中采用 PostgreSQL 是一个明智的选择。

MySQL 被 Oracle 收购之后正在逐步封闭（自 MySQL 5.5.31 以后的所有版本将不再遵循 GPL 协议），鉴于此，将来也许会选择 PostgreSQL 而不是 MySQL 作为项目的后端数据库。

Go 语言实现的支持 PostgreSQL 的驱动也很多，因为国外很多人在开发中使用了这个数据库。

- https://github.com/lib/pq 支持 database/sql 驱动，纯 Go 语言编写的。
- https://github.com/jbarham/gopgsqldriver 支持 database/sql 驱动，纯 Go 语言编写的。
- https://github.com/lxn/go-pgsql 支持 database/sql 驱动，纯 Go 语言编写的。

在下面的示例中采用第一个驱动，因为目前使用它的人最多，在 GitHub 上也比较活跃。

数据库建表语句如下：

```
CREATE TABLE userinfo
(
      uid serial NOT NULL,
      username character varying(100) NOT NULL,
      departname character varying(500) NOT NULL,
      Created date,
      CONSTRAINT userinfo_pkey PRIMARY KEY (uid)
)
WITH (OIDS=FALSE);

CREATE TABLE userdeatail
(
      uid integer,
      intro character varying(100),
      profile character varying(100)
)
WITH(OIDS=FALSE);
```

看下面这个 Go 语言程序是如何操作数据库表进行数据的增删改查的：

仅供非商业用途或交流学习使用


```
// 示例代码 11-17
package main
import (
	"database/sql"
	"fmt"

	_ "github.com/lib/pq"
)
func main() {
	db, err := sql.Open("postgres", "user=zuolan password=zuolan dbname=test sslmode
=disable")
	checkErr(err)

	//插入数据
	stmt, err := db.Prepare("INSERT INTO userinfo(username,departname,created)
VALUES($1,$2,$3) RETURNING uid")
	checkErr(err)

	res, err := stmt.Exec("张三", "研发部门", "2017-09-09")
	checkErr(err)

	// pg 不支持这个函数，因为它没有类似 MySQL 的自增 ID
	// id, err := res.LastInsertId()
	// checkErr(err)
	// fmt.Println(id)

	var lastInsertId int
	err = db.QueryRow("INSERT INTO userinfo(username,departname,created) VALUES
($1,$2,$3) returning uid;", "zuolan", "研发部门", "2017-09-09").Scan(&lastInsertId)
	checkErr(err)
	fmt.Println("最后插入 id =", lastInsertId)


	//更新数据
	stmt, err = db.Prepare("update userinfo set username=$1 where uid=$2")
	checkErr(err)

	res, err = stmt.Exec("zuolanupdate", 1)
	checkErr(err)

	affect, err := res.RowsAffected()
	checkErr(err)

	fmt.Println(affect)

	//查询数据
```

仅供非商业用途或交流学习使用


```
        rows, err := db.Query("SELECT * FROM userinfo")
        checkErr(err)

        for rows.Next() {
                var uid int
                var username string
                var department string
                var created string
                err = rows.Scan(&uid, &username, &department, &created)
                checkErr(err)
                fmt.Println(uid)
                fmt.Println(username)
                fmt.Println(department)
                fmt.Println(created)
        }

        //删除数据
        stmt, err = db.Prepare("delete from userinfo where uid=$1")
        checkErr(err)

        res, err = stmt.Exec(1)
        checkErr(err)

        affect, err = res.RowsAffected()
        checkErr(err)

        fmt.Println(affect)

        db.Close()

}

func checkErr(err error) {
        if err != nil {
                panic(err)
        }
}
```

从上面的代码可以看到，PostgreSQL 是通过$1、$2 这种方式来指定要传递的参数的，而不是 MySQL 中的？。另外，在 sql.Open 中的 dsn 信息的格式也与 MySQL 的驱动中的 dsn 格式不一样，所以在使用时请注意它们的差异。

还有 PostgreSQL 不支持 LastInsertId 函数，因为 PostgreSQL 内部没有实现类似 MySQL 的自增 ID 返回，其他的代码几乎是一模一样的。

仅供非商业用途或交流学习使用

11.4.5 NoSQL 数据库操作

NoSQL（Not Only SQL），指的是非关系型的数据库。随着 Web 2.0 的兴起，传统的关系型数据库在应付 Web 2.0 网站，特别是超大规模和高并发的 SNS 类型的 Web 2.0 纯动态网站时已经显得力不从心，暴露了很多难以克服的问题，而非关系型的数据库则由于其本身的特点得到了非常迅速的发展。

而 Go 语言作为 21 世纪的 C 语言，对 NoSQL 的支持也是很好，目前流行的 NoSQL 主要有 Redis、MongoDB、Cassandra 和 Membase 等。这些数据库都有高性能、高并发读写等特点，目前已经广泛应用于各种应用中。

1. Redis

Redis 是一个键值对存储系统，和 Memcached 类似，它支持存储的 value 类型相对更多，包括 string（字符串）、list（链表）、set（集合）和 zset（有序集合）。

目前应用 Redis 最广泛的应该是新浪微博平台，其次还有 Facebook 收购的图片社交网站 instagram。Go 语言目前支持 Redis 的驱动如下：

- https://github.com/garyburd/redigo
- https://github.com/go-redis/redis
- https://github.com/hoisie/redis
- https://github.com/alphazero/Go-Redis
- https://github.com/simonz05/godis

推荐使用第一个，用法和前面的驱动没有太大区别，限于篇幅不再介绍。

2. MongoDB

MongoDB 是一个高性能、开源、无模式的文档型数据库，是一个介于关系型数据库和非关系型数据库之间的产品，是非关系型数据库当中功能最丰富，最像关系型数据库的。它支持的数据结构非常松散，采用的是类似 json 的 bjson 格式来存储数据的，因此可以存储比较复杂的数据类型。MongoDB 最大的特点是它支持的查询语言非常强大，其语法有点类似于面向对象的查询语言，几乎可以实现类似关系型数据库单表查询的绝大部分功能，而且还支持对数据建立索引。

目前 Go 语言支持 MongoDB 最好的驱动就是 mgo（http://labix.org/mgo），这个驱动目前最有可能成为官方的 pkg。安装 mgo 的命令为：

```
go get gopkg.in/mgo.v2
```

仅供非商业用途或交流学习使用

11.5　Go 语言使用 Cookie

Web 开发中一个很重要的议题就是如何做好用户整个浏览过程的控制，因为 HTTP 协议是无状态的，所以用户的每一次请求都是无状态的，不知道在整个 Web 操作过程中哪些连接与该用户有关。应该如何来解决这个问题呢？Web 里面经典的解决方案是 Cookie 和 Session。

Cookie 机制是一种客户端机制，把用户数据保存在客户端，而 Session 机制是一种服务器端的机制，服务器使用一种类似于散列表的结构来保存信息，每一个网站访客都会被分配给一个唯一的标识符，即 sessionID，它的存放形式无非两种：要么经过 URL 传递，要么保存在客户端的 Cookie 里。当然，也可以将 Session 保存到数据库里，这样会更安全，但效率方面会有所下降。

本节主要介绍 Go 语言使用 Cookie 的方法。

11.5.1　设置 Cookie

Go 语言中通过 net/http 包中的 SetCookie 来设置 Cookie：

```
http.SetCookie(w ResponseWriter, cookie *Cookie)
```

w 表示需要写入的 response，cookie 是一个 struct，让我们来看看对象是怎样的：

```
type Cookie struct {
      Name       string
      Value      string
      Path       string
      Domain     string
      Expires    time.Time
      RawExpires string

    // MaxAge=0 意味着没有指定 Max-Age 的值
    // MaxAge<0 意味着现在就删除 Cookie，等价于 Max-Age=0
    // MaxAge>0 意味着 Max-Age 属性存在并以秒为单位存在
      MaxAge   int
      Secure   bool
      HttpOnly bool
      Raw      string
      Unparsed []string // 未解析的 attribute-value 属性值对
}
```

下面来看一个如何设置 Cookie 的例子：

```
expiration := time.Now()
expiration = expiration.AddDate(1, 0, 0)
cookie := http.Cookie{Name: "username", Value: "zuolan", Expires: expiration}
http.SetCookie(w, &Cookie)
```

11.5.2 读取 Cookie

上面的例子演示了如何设置 Cookie 数据，这里演示如何读取 Cookie：

```
cookie, _ := r.Cookie("username")
fmt.Fprint(w, cookie)
```

还有另外一种读取方式：

```
for _, cookie := range r.Cookies() {
      fmt.Fprint(w, cookie.Name)
}
```

可以看到通过 request 获取 Cookie 非常方便。

11.6 本章小结

本章几乎把常见的网络编程的内容全部介绍了一遍，Go 语言目前最为流行的便是用在服务器端开发上，因此在网络编程这方面，Go 语言可以说有着得天独厚的优势，也拥有十分丰富的社区生态。如果读者遇到问题，大部分情况在社区中都能很容易得到解答。

第 12 章

I/O 编程

在前面的学习中，几乎每个例子都有标准库的身影，标准库为大部分程序的编写提供了必要的基础功能，本章内容根据功能分类介绍了部分标准库，主要包括“输入/输出”“文件系统”“文本与数据处理”“数学计算”以及“同步”相关的包。

12.1 输入/输出

Go 语言将 I/O 操作封装在以下几个包中。

- io：为 I/O 原语（I/O primitives）提供基本的接口，在 io 包中最重要的是两个接口——Reader 和 Writer 接口。本章所提到的各种 I/O 包，都和这两个接口有关，只要程序实现了这两个接口，它就有了 I/O 的功能。
- io/ioutil：封装一些实用的 I/O 函数，这个包主要提供了一些常用、方便的 I/O 操作函数。
- fmt：实现格式化 I/O，类似 C 语言中的 printf 和 scanf。
- bufio：实现带缓冲的 I/O。它封装于 io.Reader 和 io.Writer 对象，创建了另一个对象（Reader 和 Writer），在提供缓冲的同时实现了一些文本 I/O 的功能。

12.1.1 io：基本 I/O 接口

io 包为 I/O 原语提供了基本的接口，它主要包装了这些原语的已有实现。由于这些接口和原语以不同的实现包装了低级操作，因此除非另行通知，否则客户端不应假定它们对于并行执行是安全的。

在 io 包中最重要的是两个接口：Reader 和 Writer 接口，本章所提到的各种 io 包，都与这两个接口有关，也就是说，只要实现了这两个接口，它就有了 I/O 的功能。

1. Reader 接口

Reader 接口的定义如下：

```
type Reader interface {
    Read(p []byte) (n int, err error)
}
```

上面是 io.Reader 接口定义，只有一个 Read 方法，这个方法接受一个 byte 类型的切片，并返回两个值，一个是读入的字节数 n，另一个是错误 err。

io.Reader 接口的规则比较复杂，整理一下分为以下几点：

（1）Read 方法最多读取 len(p)字节的数据，并保存到 p；

（2）Read 方法返回读取的字节数以及任何发生的错误信息；

（3）读取的字节数 n 要满足 0 <= n <= len(p)；

（4）读取的字节数 n<len(p)时，表示读取的数据不足以填满 p，这时方法会立即返回，而不是等待更多的数据；

（5）读取过程中遇到错误，会返回读取的字节数 n 以及相应的错误 err；

（6）在底层输入流结束时，方法会返回 n>0 字节，但是遇到错误可能会中断（EOF），也可以返回 nil；

（7）在第 6 种情况下，再次调用 Read 方法的时候，肯定会返回(0, EOF)；

（8）调用 Read 方法时，如果 n>0 时，优先处理读入的数据，然后再处理错误 err，EOF 也要这样处理；

（9）Read 方法不鼓励返回 n=0 并且 err=nil 的情况（返回字节数 n 为空时不推荐把 err 设置为 nil）。

注意第 8 条，即使在读取的时候遇到错误，也应该处理已经读到的数据，因为这些已经读到的数据是正确的，如果不处理那些丢失的数据（再读一次之类），读到的数据就不完整了。

由于 Reader 接口的方法集只包含一个 Read 方法，因此，所有实现了 Read 方法的类型都实现了 io.Reader 接口，也就是说，在所有需要 io.Reader 的地方，可以传递实现了 Read()方法的类型的实例。

下面，通过具体例子来谈谈该接口的用法：

```
func ReadFrom(reader io.Reader, num int) ([]byte, error) {
      p := make([]byte, num)
      n, err := reader.Read(p)
      if n > 0 {
            return p[:n], nil
      }
      return p, err
}
```

ReadFrom 函数将 io.Reader 作为参数，也就是说，ReadFrom 可以从任意地方读取数据，只要来源实现了 io.Reader 接口。比如，可以从标准输入、文件、字符串等读取数据，示例代码如下：

```
// 从标准输入读取
data, err = ReadFrom(os.Stdin, 11)

// 从普通文件读取，其中 file 是 os.File 的实例
data, err = ReadFrom(file, 9)

// 从字符串读取
data, err = ReadFrom(strings.NewReader("from string"), 12)
```

注意，io.EOF 变量的定义：var EOF = errors.New("EOF")是 error 类型。根据 Reader 接口的说明，在 n > 0 且数据被读完的情况下，返回的 error 有可能是 EOF 也有可能是 nil。

2. Writer 接口

Writer 接口的定义如下：

```
type Writer interface {
    Write(p []byte) (n int, err error)
}
```

从上面定义位置的源码注释来看，这个方法有严格的规范要求，要想实现一个 io.Writer 接口，必须遵循以下规则：

（1）Write 方法向底层数据流写入 len(p)字节的数据，这些数据来自于切片 p；

（2）返回被写入的字节数 n，其中 0 <= n <= len(p)；

（3）如果 n<len(p)，则必须返回一些非 nil 的 err（和 Read 一样）；

（4）如果中途出现问题，也要返回非 nil 的 err；

（5）Write 方法绝对不能修改切片 p 以及其中的数据。

在上一个例子中，实现了一个函数接收一个 io.Reader 类型的参数。下面通过标准库的例子来学习。

在 fmt 标准库中，有一组函数：Fprint/Fprintf/Fprintln，它们接收一个 io.Writer 类型参数（第一个参数），也就是说它们将数据格式化输出到 io.Writer 中。以 fmt.Fprintln 为例，同时看一下 fmt.Println 函数的源码：

```
func Println(a ...interface{}) (n int, err error) {
      return Fprintln(os.Stdout, a...)
}
```

很显然，fmt.Println 会将内容输出到标准输出中。下面将详细介绍 fmt 包。

12.1.2 fmt：格式化 I/O

fmt 包实现了格式化 I/O 函数，类似于 C 语言中的 printf 和 scanf。格式占位符衍生自 C 语言，但比 C 语言更简单。

fmt 包的官方文档对 Printing 和 Scanning 有很详细的说明，这里就直接引用文档进行说明，同时附上额外的说明或例子，之后再介绍具体的函数使用。

```
type Website struct {
    Name string
}

// 打印结构体时
var site = Website{Name:"Golang"}
```

1. Print 序列函数

这里说的 Print 序列函数包括：Fprint/Fprintf/Fprintln/Sprint/Sprintf/Sprintln/Print/Printf/Println。之所以将它们放在一起介绍，是因为它们的使用方式类似，参数含义也类似。

在一般情况下，将 Fprint/Fprintf/Fprintln 归为一类；Sprint/Sprintf/Sprintln 归为一类；Print/Printf/Println 归为另一类。其中，Print/Printf/Println 会调用相应的 F 开头的函数，如：

```
func Print(a ...interface{}) (n int, err error) {
      return Fprint(os.Stdout, a...)
}
```

Fprint/Fprintf/Fprintln 函数的第一个参数接收一个 io.Writer 类型，会将内容输出到 io.Writer 中去。而 Print/Printf/Println 函数是将内容输出到标准输出中，因此，直接调用 F 类函数做这件事，并将 os.Stdout 作为第一个参数传入。

Sprint/Sprintf/Sprintln 只是格式化内容为 string 类型，而并不输出到某处，需要格式化字符串并返回时，可以用 Stdout 函数。

在这三组函数中，Sprintf/Fprintf/Printf 函数通过指定的格式输出或格式化内容；Sprint/Fprint/Print 函数只是使用默认的格式输出或格式化内容；Sprintln/Fprintln/Println 函数使用默认的格式输出或格式化内容，同时会在最后加上"换行符"。

Print 序列函数的最后一个参数都是 a ...interface{}这种不定参数。对于 Sprintf/Fprintf/Printf 序列，这个不定参数的实参个数应该和 formt 参数的占位符个数一致，否则会出现格式化错误；而对于其他函数，当不定参数的实参个数为多个时，它们之间会直接（对于 Sprint/Fprint/Print）或通过" "（空格）（对于 Sprintln/Fprintln/Println）连接起来（对于 Sprint/Fprint/Print，当两个参数都不是字符串时，会自动添加一个空格，否则不会添加）。利用这一点，我们可以做如下事情：

```
result1 := fmt.Sprintln("example.com", 2017)
result2 := fmt.Sprint("example.com", 2017)
```

result1 的值是 example.com 2017，result2 的值是 example.com2017，这起到了连接字符串的作用，而不需要通过 strconv.Itoa()转换。

Print 序列函数用得较多，而且也易于使用（可能需要掌握一些常用的占位符用法），接下来结合 fmt 包中几个相关的接口来掌握更多关于 Print 的内容。

2. Stringer 接口

Stringer 接口的定义如下：

```
type Stringer interface {
    String() string
}
```

根据 Go 语言中实现接口的定义，一个类型只要有 String() string 方法，就说它实现了 Stringer 接口。而在本节开始已经说过，如果格式化输出某种类型的值，只要它实现了 String()方法，那么就会调用 String()方法进行处理。

定义如下 struct：

```
type Person struct {
      Name string
      Age  int
      Sex  int
}
```

给 Person 实现 String 方法，下面输出 Person 的实例：

```
p := &Person{"张三", 28, 0}
fmt.Println(p)
/*
返回:
&{张三 28 0}
*/
```

接下来，为 Person 增加 String 方法：

```
func (this *Person) String() string {
      buffer := bytes.NewBufferString("这是")
      buffer.WriteString(this.Name + ", ")
      if this.Sex == 0 {
            buffer.WriteString("他")
      } else {
            buffer.WriteString("她")
      }
      buffer.WriteString("今年")
      buffer.WriteString(strconv.Itoa(this.Age))
      buffer.WriteString("岁。")
      return buffer.String()
}
```

这个时候运行：

```
p := &Person{"zuolan", 28, 0}
fmt.Println(p)
/*
返回:
这是张三, 他今年 28 岁。
*/
```

可见，Stringer 接口和 Java 中的 ToString 方法类似。

3. Formatter 接口

Formatter 接口的定义如下：

```
type Formatter interface {
    Format(f State, c rune)
}
```

Formatter 接口由带有定制的格式化符的值所实现。Format 的实现可调用 Sprintf 或 Fprintf(f)等函数来生成其输出。

也就是说，通过实现 Formatter 接口可以做到自定义输出格式（自定义占位符）。接着上面的例子，我们为 Person 增加一个方法：

```
func (this *Person) Format(f fmt.State, c rune) {
      if c == 'L' {
            f.Write([]byte(this.String()))
            f.Write([]byte("Person 有三个字段。"))
      } else {
            // 没有此句，会导致 fmt.Printf("%s", p) 什么也不输出
          f.Write([]byte(fmt.Sprintln(this.String())))
      }
}
```

这样，Person 便实现了 Formatter 接口，这时再运行：

```
p := &Person{"zuolan", 28, 0}
fmt.Printf("%L", p)
/*
返回：
这是张三，他今年 28 岁。Person 有三个字段。
*/
```

这里需要解释以下几点：

- fmt.State 是一个接口。由于 Format 方法是被 fmt 包调用的，它内部会实例化一个 fmt.State 接口的实例，我们不需要关心该接口。
- 可以实现自定义占位符，同时 fmt 包中和类型相对应的预定义占位符会无效。因此例子中 Format 的实现加上了 else 子句。
- 实现了 Formatter 接口，相应的 Stringer 接口不起作用。但实现了 Formatter 接口的类型应该实现 Stringer 接口，这样方便在 Format 方法中调用 String()方法。就像本例的做法。
- Format 方法的第二个参数是占位符中%后的字母（有精度和宽度会被忽略，只保留字母）。

在一般情况下，不需要实现 Formatter 接口。如果对 Formatter 接口的实现感兴趣，可以看看标准库 math/big 包中 Int 类型的 Formatter 接口实现。

State 接口相关的说明如下：

```
type State interface {
    // Write 函数用于打印出已格式化的输出
    Write(b []byte) (ret int, err error)
    // Width 返回宽度选项的值以及它是否已被设置
    Width() (wid int, ok bool)
    // Precision 返回精度选项的值以及它是否已被设置
    Precision() (prec int, ok bool)
    // Flag 返回标记 c（一个字符）是否已被设置
    Flag(c int) bool
}
```

fmt 包的 print.go 文件中的 type pp struct 实现了 State 接口。由于 State 接口有 Write 方法，因此，实现了 State 接口的类型必然实现了 io.Writer 接口。

4. GoStringer 接口

GoStringer 接口的定义如下：

```
type GoStringer interface {
    GoString() string
}
```

该接口定义了类型的 Go 语言语法格式。用于打印（Printf）格式化占位符为%#v 的值。继续用前面的例子演示，执行以下代码：

```
p := &Person{"张三", 28, 0}
fmt.Printf("%#v", p)
/*
返回：
&main.Person{Name:"张三", Age:28, Sex:0}
*/
```

接着为 Person 增加方法：

```
func (this *Person) GoString() string {
      return "&Person{姓名："+this.Name+", 年龄："+strconv.Itoa(this.Age)+", 性别："
"+strconv.Itoa(this.Sex)+"}"
}
```

这个时候再执行：

```
p := &Person{"张三", 28, 0}
fmt.Printf("%#v", p)
/*
返回：
&Person{姓名：张三，年龄：28，性别：0}
*/
```

在一般情况下不需要实现该接口。

5. Scan 序列函数

该序列函数和 Print 序列函数相对应，包括：Fscan/Fscanf/Fscanln/Sscan/Sscanf/Sscanln/Scan/Scanf/Scanln。

在一般情况下将 Fscan/Fscanf/Fscanln 归为一类；Sscan/Sscanf/Sscanln 归为一类；Scan/Scanf/Scanln 归为另一类。其中，Scan/Scanf/Scanln 会调用相应的 F 开头的那一类函数，如：

```
func Scan(a ...interface{}) (n int, err error) {
      return Fscan(os.Stdin, a...)
}
```

Fscan/Fscanf/Fscanln 函数的第一个参数接收一个 io.Reader 类型，从其中读取内容并赋值给相应的实参。而 Scan/Scanf/Scanln 正是从标准输入获取内容，因此，直接调用 F 类函数做这件事，并将 os.Stdin 作为第一个参数传入。

Sscan/Sscanf/Sscanln 则直接从字符串中获取内容。

对于 Scan/Scanf/Scanln 三个函数的区别，通过例子来说明，为了方便讲解，使用 Sscan/Sscanf/Sscanln 这组函数。

（1）Scan/FScan/Sscan

```
var ( name string age int ) n, _ := fmt.Sscan("张三 28", &name, &age)
// 可以将"张三 28"中的空格换成"\n"试试
// n, _ := fmt.Sscan("张三\n28", &name, &age)
fmt.Println(n, name, age)
// 返回：2 张三 28
```

不管“张三 28”是用空格分隔还是用“\n”分隔，输出一样，也就是说，Scan/FScan/Sscan 这组函数将连续由空格分隔的值存储为连续的实参（换行符也记为空格）。

（2）Scanf/FScanf/Sscanf

```
var ( name string age int ) n, _ := fmt.Sscanf("张三 28", "%s%d", &name, &age)
// 可以将"张三 28"中的空格换成"\n"试试
// n, _ := fmt.Sscanf("张三\n28", "%s%d", &name, &age)
fmt.Println(n, name, age)
// 返回：2 张三 28
```

如果将“空格”分隔改为“\n”分隔，则输出为：1 zuolan 0，可见，Scanf/FScanf/Sscanf 这组函数将连续由空格分隔的值存储为连续的实参，其格式由 format 决定，换行符处停止扫描（Scan）。

（3）Scanln/FScanln/Sscanln

```
var ( name string age int ) n, _ := fmt.Sscanln("张三 28", &name, &age)
// 可以将"张三 28"中的空格换成"\n"试试
// n, _ := fmt.Sscanln("张三\n28", &name, &age)
fmt.Println(n, name, age)
// 返回：2 张三 28
```

Scanln/FScanln/Sscanln 表现和上一组一样，遇到“\n”停止（对于 Scanln，表示从标准输入获取内容，最后需要回车）。在一般情况下使用 Scan/Scanf/Scanln 这组函数。

12.1.3 文本处理

Go 语言标准库中有几个包专门用于处理文本。

strings：字符串操作，这个包提供了很多操作字符串的简单函数，通常一般的字符串操作需求都可以在这个包中找到。

- vbytes：byte slice 便利操作，在 Go 语言中 string 是内置类型，同时它与普通的 slice 类型有着相似的性质，均可以进行切片（slice）操作。
- strconv：字符串和基本数据类型之间转换，因为在 Go 语言中，没有隐式类型转换，字符串类型和 int、float、bool 等类型之间不能直接使用基本数据类型那套方法转换，需要使用这个包的方法实现转换。
- regexp：正则表达式，谈到字符串必然离不开正则表达式，这个包提供了正则表达式的功能。
- unicode：Unicode 编码、UTF-8/16 编码。

12.2 文件系统

I/O 操作只是对文件的读写，os 包还提供了访问文件的其他一些属性，例如权限、大小等。可以进行文件夹的新建、删除、移动、复制、重命名等。

os 包提供了平台无关的操作系统功能接口，其错误处理采用的是 UNIX 风格的设计。

12.2.1 os：系统功能实现

os 包提供了与平台无关的操作系统功能接口。尽管错误处理是 Go 语言风格，但设计是 UNIX 风格，所以，失败的调用会返回 error 而非错误码。通常 error 会包含更多信息，例如，如果使用一个文件名的调用（如 Open、Stat）失败了，打印错误时会包含该文件名，错误类型将为*PathError，错误内部可以通过解包（解析该引用值的具体信息，返回类型的内部包含对应错误的详细内容）获得更多信息。

os 包的接口规定实现在所有操作系统中都是一致的。有一些某个系统特定的功能，需要使用 syscall 获取。实际上，os 依赖于 syscall。在实际编程中，我们应该总是优先使用 os 中提供的功能，而不是 syscall。

下面是一个简单的例子，打开一个文件并从中读取一些数据：

```
file, err := os.Open("file.go") // 用于读取访问。
if err != nil {
        log.Fatal(err)
}
```

如果打开失败，错误字符串是能描述原因的，例如：open file.go: no such file or directory，不像 C 语言，需要额外的函数（或宏）来解释错误码。

详尽的 os 包与 io 包操作都在前后介绍过（本章第一节和下一节分别介绍 io 包和 os 包的内容），此处不再赘述。

12.2.2 path：兼容路径操作

path/filepath 包涉及路径操作时，路径分隔符使用 os.PathSeparator，不同系统的路径表示方式有所不同，比如 UNIX 和 Windows 差别很大。Path 包能够处理所有的文件路径，不管是什么系统。注意，路径操作函数并不会校验路径是否真实存在。

1. 解析路径名字符串

Dir()和 Base()函数将一个路径名字符串分解成目录和文件名两部分（这些函数与 UNIX 中的 dirname 和 basename 命令类似，但如果路径以/结尾，Dir()的行为和 dirname 不太一致）。

```
func Dir(path string) string
func Base(path string) string
```

Dir()返回路径中除去最后一个路径元素的部分，即该路径最后一个元素所在的目录。在使用 Split 去掉最后一个元素后，会简化路径并去掉末尾的斜杠。如果路径是空字符串，会返回“.”，如果路径是“/”开头并且后面是路径则返回“/”，其他任何情况下都不会返回以“/”结尾的路径。

Base()函数返回路径的最后一个元素。在提取元素前会去掉末尾的“/”。如果路径是空，会返回“.”；如果路径是只有一个“/”构成的，会返回“/”。

简单来说，给定路径名/home/zuolan/example.go，Dir()返回/home/zuolan，而 Base()返回 example.go。

如果给定路径名/home/zuolan/example/，Dir()返回/home/zuolan/example（这与 UNIX 中的 dirname 不一致，dirname 会返回/home/zuolan，如果要和 dirname 一样，应该在调用 Dir()之前先将末尾的“/”去掉），而 Base()返回 example。

此外，Ext()可以获得路径中文件名的扩展名，例如：func Ext(path string) string

Ext 函数返回 path 文件扩展名。扩展名是路径中最后一个从“.”开始的部分，包括“.”。如果该元素没有“.”会返回空字符串。

2. 相对路径和绝对路径

每个进程都会有当前工作目录，一般的相对路径，就是针对进程当前工作目录而言的。当然，可以针对某个目录指定相对路径。

绝对路径，在 UNIX 中以“/”开始；在 Windows 中以某个盘符开始，比如 C:\Program Files 等，IsAbs()函数可以判断返回的路径是否是一个绝对路径：

```
func IsAbs(path string) bool
```

而函数 Abs()返回 path 代表的绝对路径，如果输入参数 path 不是绝对路径，会加入当前工作目录以使之成为绝对路径：

```
func Abs(path string) (string, error)
```

因为硬链接的存在，不能保证返回的绝对路径是唯一指向该地址的绝对路径。在 os.Getwd 出错时，Abs()会返回该错误，如果路径名长度超过系统限制，也会报错。

Rel()函数返回一个相对路径，将 basepath 和该路径用路径分隔符连起来的新路径在词法上等价于 targpath：

```
func Rel(basepath, targpath string) (string, error)
```

也就是说，Join(basepath, Rel(basepath, targpath))等价于 targpath。如果成功执行，返回值总是相对于 basepath 的，即使 basepath 和 targpath 没有共享的路径元素。如果两个参数一个是相对路径而另一个是绝对路径，或者 targpath 无法表示为相对于 basepath 的路径，将返回错误。

```
fmt.Println(filepath.Rel("/home/zuolan/example", "/home/zuolan/example/src/logic/t
opic.go"))
fmt.Println(filepath.Rel("/home/zuolan/example", "/data/example"))
/*
返回：
src/logic/topic.go <nil>
../../../data/example <nil>
*/
```

3. 路径的切分和拼接

对于一个常规文件路径，可以通过 Split 函数得到它的目录路径和文件名：

```
func Split(path string) (dir, file string)
```

Split()函数根据最后一个路径分隔符将路径 path 分隔为目录和文件名两部分（dir 和 file）。如果路径中没有路径分隔符，函数返回值 dir 为空字符串，file 等于 path；反之，如果路径中最后一个字符是“/”，则 dir 等于 path，file 为空字符串。返回值满足 path == dir+file。dir 非空时，最后一个字符总是“/”。

```
// dir == /home/zuolan/, file == example
filepath.Split("/home/zuolan/example")

// dir == /home/zuolan/example/, file == ""
filepath.Split("/home/zuolan/example/")

// dir == "", file == example
filepath.Split("example")
```

相对路径到绝对路径的转变，需要经过路径的拼接。Join 用于将多个路径拼接起来，会根据情况添加路径分隔符：

```
func Join(elem ...string) string
```

Join()函数可以将任意数量的路径元素放入一个单一路径里，会根据需要添加路径分隔符。结果是经过 Clean 的，所有的空字符串元素会被忽略。对于拼接路径的需求，应该总是使用 Join 函数来处理。

有时，需要分割 PATH 或 GOPATH 之类的环境变量（这些路径被特定于 OS 的列表分隔符连接起来），filepath.SplitList 就是这个用途：

```
func SplitList(path string) []string
```

注意，与 strings.Split 函数的不同之处在于：对于空路径，SplitList 返回[]string{}，而 strings.Split 返回[]string{""}。SplitList 内部调用的是 strings.Split。

4．规整化路径

Clean()函数通过单纯的词法操作返回和 path 代表同一地址的最短路径：

```
func Clean(path string) string
```

它会不断地依次应用如下的规则，直到不能再进行任何处理：

（1）将连续的多个路径分隔符替换为单个路径分隔符。

（2）剔除每一个“.”路径名元素（代表当前目录）。

（3）剔除每一个路径内的“..”路径名元素（代表父目录）和它前面的非“..”路径名元素。

（4）剔除开始于根路径的“..”路径名元素，即将路径开始处的“/..”替换为“/”（假设路径分隔符是“/”）。

返回的路径只有其代表一个根地址时才以路径分隔符结尾，如 UNIX 的/或 Windows 的 C:\。

如果处理的结果是空字符串，Clean 会返回“.”，代表当前路径。

5. 文件路径匹配

Match()指示 name 是否和 shell 的文件模式匹配。模式语法如下：

```
func Match(pattern, name string) (matched bool, err error)

pattern:
      { term }
term:
      '*'             匹配 0 或多个非路径分隔符的字符
      '?'             匹配 1 个非路径分隔符的字符
      '[' [ '^' ] { character-range } ']'
                      字符组（必须非空）
      c               匹配字符 c（c != '*', '?', '\\', '['）
      '\\' c          匹配字符 c
character-range:
      c               匹配字符 c（c != '\\', '-', ']'）
      '\\' c          匹配字符 c
      lo '-' hi       匹配区间[lo, hi]内的字符
```

匹配要求 pattern 必须和 name 全匹配上，不只是子串。在 Windows 下转义字符被禁用。Match()函数很少使用，标准库没有用到这个函数。而 Glob 函数在模板标准库中被用到了：

```
func Glob(pattern string) (matches []string, err error)
```

Glob()函数返回所有匹配了模式字符串 pattern 的文件列表或者 nil（如果没有匹配的文件）。pattern 的语法和 Match 函数相同。pattern 可以描述多层的名字，如/usr/*/bin/ed（假设路径分隔符是“/”）。

注意，Glob()会忽略任何文件系统相关的错误，如读目录引发的 I/O 错误。唯一的错误和 Match()一样，在 pattern 不合法时，返回 filepath.ErrBadPattern。返回的结果是根据文件名字典顺序进行了排序的。

Glob()的常见用法，是读取某个目录下所有的文件，比如写单元测试时，读取 testdata 目录下的所有测试数据：

```
filepath.Glob("testdata/*.input")
```

6. 遍历目录

在介绍 os 时，讲解了读取目录的方法，并给出了一个遍历目录的示例。在 filepath 中，提供了 Walk()函数，用于遍历目录树。

```
func Walk(root string, walkFn WalkFunc) error
```

Walk()函数会遍历 root 指定的目录下的文件树，对每一个该文件树中的目录和文件都会调用 walkFn，包括 root 自身。所有访问文件/目录时遇到的错误都会传递给 walkFn 过滤。文件是按字典顺序遍历的，这让输出更漂亮，但也导致处理非常大的目录时效率会降低。Walk()函数不会遍历文件树中的符号链接（快捷方式）文件包含的路径。

walkFn 的类型 WalkFunc 的定义如下：

```
type WalkFunc func(path string, info os.FileInfo, err error) error
```

Walk()函数对每一个文件/目录都会调用 WalkFunc()函数类型值。调用时 path 参数会包含 Walk()的 root 参数作为前缀，也就是说，如果 Walk()函数的 root 为“dir”，该目录下有文件“a”，将会使用“dir/a”作为调用 walkFn 的参数。walkFn 参数被调用时的 info 参数是 path 指定的地址（文件/目录）的文件信息，类型为 os.FileInfo。

如果遍历 path 指定的文件或目录时出现了问题，传入的参数 err 会描述该问题，WalkFunc 类型函数可以决定如何去处理该错误（Walk 函数将不会深入该目录）；如果该函数返回一个错误，Walk 函数的执行会中止；只有一个例外，如果 Walk 的 walkFn 返回值是 SkipDir，将会跳过该目录的内容而 Walk 函数照常执行处理下一个文件。

最后总结一下，在 UNIX 中，路径的分隔符是“/”，但 Windows 中的是“\”。在使用 path 包时，应该总是使用“/”，不论什么系统。

path 包中提供的函数，filepath 都有提供，功能类似，但实现不同。一般应该使用 filepath 包，而不是 path 包。

12.3　数据结构与算法

数学方面的包主要是 math——实现数学函数计算，math 的子包包含以下这些包：

- math/big：大整数的高精度计算实现。
- math/cmplx：复数基本函数操作。
- math/rand：伪随机数生成器。

Go 语言在数据结构方面主要有三大相关的标准包：

- sort：包含基本的排序方法，支持切片数据排序以及用户自定义数据集合排序。
- index/suffixary：该包实现了后缀数组相关算法以支持许多常见的字符串操作。
- container：该包实现了三个复杂的数据结构，即堆、链表和环。这个包表示使用这三个数据结构的时候不需要再费心从头开始写算法了。

12.3.1 排序

Go 语言中实现了四种基本排序算法：插入排序、归并排序、堆排序和快速排序，但是这四种排序方法是不公开的，它们只被 sort 包内部使用。

在对数据集合排序时不必考虑应当选择哪一种排序方法，只要实现了 sort.Interface 定义的三个方法：获取数据集合长度的 Len()方法、比较两个元素大小的 Less()方法和交换两个元素位置的 Swap()方法，就可以顺利对数据集合进行排序。

sort 包会根据实际数据自动选择高效的排序算法。除此之外，为了方便对常用数据类型的操作，sort 包提供了对[]int 切片、[]float64 切片和[]string 切片的完整支持，主要包括：对基本数据类型切片的排序支持；基本数据元素查找；判断基本数据类型切片是否已经排好序；对排好序的数据集合逆序。

1. 数据集合排序

前面已经提到过，对数据集合（包括自定义数据类型的集合）排序需要实现 sort.Interface 接口的三个方法，我们看一下该接口的定义：

```
type Interface interface {
    // 获取数据集合元素个数
    Len() int
    // 如果 i 索引的数据小于 j 所有的数据，返回 true，不会调用
    // 下面的 Swap()，即数据升序排序
    Less(i, j int) bool
    // 交换 i 和 j 索引的两个元素的位置
    Swap(i, j int)
}
```

数据集合实现了三个方法后，即可调用该包的 Sort()方法进行排序。Sort()方法的定义

如下：

```
func Sort(data Interface)
```

Sort()方法唯一的参数就是待排序的数据集合。

该包还提供了一个方法可以判断数据集合是否已经排好顺序，该方法的内部实现依赖于我们自己实现的 Len()和 Less()方法：

```
func IsSorted(data Interface) bool {
    n := data.Len()
    for i := n - 1; i > 0; i-- {
        if data.Less(i, i-1) {
            return false
        }
    }
    return true
}
```

下面是一个使用 sort 包对学生成绩排序的示例：

```
// 示例代码 12-1
package main

import (
      "fmt"
      "sort"
)

// 学生成绩结构体
type StuScore struct {
     // 姓名
    name  string
    // 成绩
    score int
}

type StuScores []StuScore

// Len(): 人数
func (s StuScores) Len() int {
      return len(s)
}

// Less(): 成绩将由低到高排序
func (s StuScores) Less(i, j int) bool {
      return s[i].score < s[j].score
}
```

```
// Swap(): 排序
func (s StuScores) Swap(i, j int) {
      s[i], s[j] = s[j], s[i]
}

func main() {
    stus := StuScores{
                {"张三", 95},
                {"李四", 91},
                {"赵五", 96},
                {"王六", 90}}

    fmt.Println("======默认======")
    // 原始顺序
    for _, v := range stus {
        fmt.Println(v.name, ":", v.score)
    }
    fmt.Println()
    // StuScores 已经实现了 sort.Interface 接口
    sort.Sort(stus)

    fmt.Println("=====排序之后=====")
     // 排好序后的结构
    for _, v := range stus {
        fmt.Println(v.name, ": ", v.score)
    }

    // 判断是否已经排好顺序，将会打印 true
    fmt.Println("是否已经排序？", sort.IsSorted(stus))
}
/*
返回:
======默认======
张三: 95
李四: 91
赵五: 96
王六: 90

=====排序之后=====
王六 : 90
李四 : 91
张三 : 95
赵五 : 96
是否已经排序？ true
*/
```

该示例程序的自定义类型 StuScores 实现了 sort.Interface 接口，所以可以将其对象作为 sort.Sort()和 sort.IsSorted()的参数传入。

该示例实现的是升序排序，如果要得到降序排序结果，其实只要修改 Less()函数即可：

```
// Less():成绩降序排序，只将小于号修改为大于号
func (s StuScores) Less(i, j int) bool {
      return s[i].score > s[j].score
}
```

此外，sort 包提供了 Reverse()方法，可以允许将数据按 Less()定义的排序方式逆序排序，而不必修改 Less()代码，方法定义如下：

```
func Reverse(data Interface) Interface
```

可以看到Reverse()返回的一个sort.Interface接口类型，整个Reverse()的内部实现如下：

```
// 定义了一个 reverse 结构类型，嵌入 Interface 接口
type reverse struct {
    Interface
}

// reverse 结构类型的 Less()方法拥有嵌入的 Less()方法相反的行为
// Len()和 Swap()方法则会保持嵌入类型的方法行为
func (r reverse) Less(i, j int) bool {
    return r.Interface.Less(j, i)
}

// 返回新的实现 Interface 接口的数据类型
func Reverse(data Interface) Interface {
    return &reverse{data}
}
```

了解内部原理后，可以在学生成绩排序示例中使用 Reverse()来实现成绩升序排序：

```
sort.Sort(sort.Reverse(stus))
 for _, v := range stus {
    fmt.Println(v.name, ":", v.score)
}
```

最后一个方法：Search()，该方法使用“二分查找”算法来搜索某指定切片[0:n]，并返回能够使 f(i)=true 的最小的 i（0<=i<n）值，并且会假定，如果 f(i)=true，则 f(i+1)=true，即对于切片[0:n]，i 之前的切片元素会使 f()函数返回 false，i 及 i 之后的元素会使 f()函数返回 true。

但是，当在切片中无法找到 f(i)=true 的 i 时（此时切片元素都不能使 f()函数返回 true），Search()方法会返回 n。

```
func Search(n int, f func(int) bool) int
```

Search()函数的一个常用方式是搜索元素 x 是否在已经升序排好的切片 s 中：

```
x := 11
s := []int{3, 6, 8, 11, 45} // 注意已经升序排序
pos := sort.Search(len(s), func(i int) bool { return s[i] >= x })
if pos < len(s) && s[pos] == x {
    fmt.Println(x, "在 s 中的位置为：", pos)
} else {
    fmt.Println("s 不包含元素", x)
}
```

官方文档还给出了一个猜数字的小程序：

```
func GuessingGame() {
    var s string
    fmt.Printf("选择一个从 0 到 100 的整数。\n")
    answer := sort.Search(100, func(i int) bool {
        fmt.Printf("这个数字是否小于等于 %d ？", i)
        fmt.Scanf("%s", &s)
        return s != "" && s[0] == 'y'
    })
    fmt.Printf("你选的数字是 %d 吧。\n", answer)
}
```

2. 切片排序

前面已经提到，sort 包原生支持[]int、[]float64 和[]string 三种内建数据类型切片的排序操作，即我们不必自己实现相关的 Len()、Less()和 Swap()方法。

1）[]int 排序

由于[]int 切片排序内部实现及使用方法与[]float64 和[]string 类似，所以只详细描述该部分。sort 包定义了一个 IntSlice 类型，并且实现了 sort.Interface 接口：

```
type IntSlice []int
func (p IntSlice) Len() int           { return len(p) }
func (p IntSlice) Less(i, j int) bool { return p[i] < p[j] }
func (p IntSlice) Swap(i, j int)      { p[i], p[j] = p[j], p[i] }
// IntSlice 类型定义了 Sort()方法，包装了 sort.Sort()函数
func (p IntSlice) Sort() { Sort(p) }
// IntSlice 类型定义了 SearchInts()方法，包装了 SearchInts()函数
func (p IntSlice) Search(x int) int { return SearchInts(p, x) }
```

提供的 sort.Ints()方法使用了该 IntSlice 类型：

```
func Ints(a []int) { Sort(IntSlice(a)) }
```

所以，对[]int 切片排序时更常使用 sort.Ints()，而不是直接使用 IntSlice 类型：

```
s := []int{5, 2, 6, 3, 1, 4} // 未排序的切片数据
sort.Ints(s)
fmt.Println(s) // 将会输出[1 2 3 4 5 6]
```

如果要使用降序排序，显然要用前面提到的 Reverse()方法：

```
s := []int{5, 2, 6, 3, 1, 4} // 未排序的切片数据
sort.Sort(sort.Reverse(sort.IntSlice(s)))
fmt.Println(s) // 将会输出[6 5 4 3 2 1]
```

如果要查找整数 x 在切片 a 中的位置，相对于前面提到的 Search()方法，sort 包提供了 SearchInts():

```
func SearchInts(a []int, x int) int
```

注意，SearchInts()的使用条件为切片 a 已经升序排序。

```
s := []int{5, 2, 6, 3, 1, 4} // 未排序的切片数据
sort.Ints(s) // 排序后的 s 为[1 2 3 4 5 6]
fmt.Println(sort.SearchInts(s, 3)) // 将会输出 2
```

2）[]float64 排序

实现与 Ints 类似，只看一下其内部实现：

```
type Float64Slice []float64

func (p Float64Slice) Len() int           { return len(p) }
func (p Float64Slice) Less(i, j int) bool { return p[i] < p[j] || isNaN(p[i])
&& !isNaN(p[j]) }
func (p Float64Slice) Swap(i, j int)      { p[i], p[j] = p[j], p[i] }
func (p Float64Slice) Sort() { Sort(p) }
func (p Float64Slice) Search(x float64) int { return SearchFloat64s(p, x) }
```

与 Sort()、IsSorted()、Search()相对应的三个方法：

```
func Float64s(a []float64)
func Float64sAreSorted(a []float64) bool
func SearchFloat64s(a []float64, x float64) int
```

要说明一下的是，在上面 Float64Slice 类型定义的 Less 方法中，有一个内部函数 isNaN()。isNaN()与 math 包中 IsNaN()实现完全相同，sort 包之所以不使用 math.IsNaN()，完全是基于包依赖性的考虑，应当看到，sort 包的实现不依赖于其他任何包。

3）[]string 排序

两个 string 对象之间的大小比较是基于“字典序”的。实现与 Ints 类似，只看一下其内部实现：

```
type StringSlice []string

func (p StringSlice) Len() int           { return len(p) }
func (p StringSlice) Less(i, j int) bool { return p[i] < p[j] }
```

```
func (p StringSlice) Swap(i, j int)      { p[i], p[j] = p[j], p[i] }
func (p StringSlice) Sort() { Sort(p) }
func (p StringSlice) Search(x string) int { return SearchStrings(p, x) }
```

与 Sort()、IsSorted()、Search()相对应的三个方法：

```
func Strings(a []string)
func StringsAreSorted(a []string) bool
func SearchStrings(a []string, x string) int
```

12.3.2 container

该包实现了三个复杂的数据结构：堆、链表和环，这个包表示使用这三个数据结构的时候不需要再费心从头开始写算法了。

1. 堆

这里的堆使用的数据结构是最小二叉树，即根节点比左边子树和右边子树的所有值都小。Go 语言的堆包只是实现了一个接口，我们看下它的定义：

```
type Interface interface {
      sort.Interface
      Push(x interface{}) // 添加一个元素 x 并返回 Len()
      Pop() interface{}   // 移除并返回元素长度 Len() - 1
}
```

可以看出，这个堆结构继承自 sort.Interface，回顾一下 sort.Interface，它需要实现三个方法：

- Len() int
- Less(i, j int) bool
- Swap(i, j int)

加上堆接口定义的两个方法：

- Push(x interface{})
- Pop() interface{}

就是说你定义了一个堆，就要实现五个方法，直接以 package doc 中的 example 为例来看一下：

```
type IntHeap []int

func (h IntHeap) Len() int           { return len(h) }
func (h IntHeap) Less(i, j int) bool { return h[i] < h[j] }
```

```
func (h IntHeap) Swap(i, j int)      { h[i], h[j] = h[j], h[i] }

func (h *IntHeap) Push(x interface{}) {
      *h = append(*h, x.(int))
}

func (h *IntHeap) Pop() interface{} {
      old := *h
      n := len(old)
      x := old[n-1]
      *h = old[0 : n-1]
      return x
}
```

那么 IntHeap 实现了这个堆结构，我们就可以使用堆的方法来对它进行操作：

```
h := &IntHeap{2, 1, 5}
heap.Init(h)
heap.Push(h, 3)
heap.Pop(h)
```

具体说一下内部实现，使用最小堆，索引排序从根节点开始，然后左子树，右子树的顺序方式。内部实现了 down 和 up 分别表示对堆中的某个元素向下保证最小堆和向上保证最小堆。

当往堆中插入一个元素的时候，这个元素插入到最右子树的最后一个节点中，然后调用 up 向上保证最小堆。

当要从堆中推出一个元素的时候，先把这个元素和右子树最后一个节点对换，然后弹出最后一个节点，再对 root 调用 down，向下保证最小堆。

2. 链表

链表就是一个有 prev 和 next 指针的数组了，它维护两个 type，注意，这里不是接口。

```
type Element struct {
      next, prev *Element  // 上一个元素和下一个元素
      list *List  // 元素所在链表
      Value interface{}  // 元素
}

type List struct {
      root Element  // 链表的根元素
      len  int      // 链表的长度
}
```

链表是先创建 list，然后往 list 中插入值，list 就在内部创建一个 Element，并在内部

设置好 Element 的 next、prev 等。具体可以看下面例子，使用堆接口构建的整数堆：

```
// 示例代码 12-2
package main

import (
      "container/list"
      "fmt"
)

func main() {
    list := list.New()
    list.PushBack(1)
    list.PushBack(2)

    fmt.Printf("长度: %v\n", list.Len());
    fmt.Printf("第一个元素: %#v\n", list.Front());
    fmt.Printf("第二个元素: %#v\n", list.Front().Next());
}
/*
返回:
长度: 2
第一个元素: &list.Element{next:(*list.Element)(0x2081be1b0), prev:(*list.Element)(0
x2081be150), list:(*list.List)(0x2081be150), Value:1}
第二个元素: &list.Element{next:(*list.Element)(0x2081be150), prev:(*list.Element)(0
x2081be180), list:(*list.List)(0x2081be150), Value:2}
*/
```

list 对应的方法如下：

```
type Element
    func (e *Element) Next() *Element
    func (e *Element) Prev() *Element
type List
    func New() *List
    func (l *List) Back() *Element   // 最后一个元素
    func (l *List) Front() *Element  // 第一个元素
    func (l *List) Init() *List  // 链表初始化
    // 在某个元素后插入
    func (l *List) InsertAfter(v interface{}, mark *Element) *Element
    // 在某个元素前插入
    func (l *List) InsertBefore(v interface{}, mark *Element) *Element
    func (l *List) Len() int // 链表长度
    func (l *List) MoveAfter(e, mark *Element)  // 把 e 元素移动到 mark 之后
    func (l *List) MoveBefore(e, mark *Element)  // 把 e 元素移动到 mark 之前
    func (l *List) MoveToBack(e *Element) // 把 e 元素移动到队列最后
    func (l *List) MoveToFront(e *Element) // 把 e 元素移动到队列最头部
    func (l *List) PushBack(v interface{}) *Element  // 在队列最后插入元素
    func (l *List) PushBackList(other *List)  // 在队列最后插入接上新队列
```

```
func (l *List) PushFront(v interface{}) *Element  // 在队列头部插入元素
func (l *List) PushFrontList(other *List) // 在队列头部插入接上新队列
func (l *List) Remove(e *Element) interface{} // 删除某个元素
```

3. 环

环的结构有点特殊，环的尾部就是头部，所以每个元素实际上就可以代表自身的这个环。它不需要像 list 一样保持 list 和 element 两个结构，只需保持一个结构就行。

```
type Ring struct {
      next, prev *Ring
      Value      interface{}
}
```

初始化环的时候，需要定义好环的大小，然后对环的每个元素进行赋值。环还提供一个 Do 方法，能遍历一遍环，对每个元素执行一个功能。看下面的例子，同样是使用堆接口构建的整数堆：

```
// 示例代码 12-3
package main

import (
      "container/ring"
      "fmt"
)

func main() {
    ring := ring.New(3)

    for i := 1; i <= 3; i++ {
        ring.Value = i
        ring = ring.Next()
    }

    // 计算 1+2+3
    s := 0
    ring.Do(func(p interface{}){
        s += p.(int)
    })
    fmt.Println("和为：", s)
}
/*
返回：
和为：6
*/
```

ring 提供的方法如下：

```
type Ring
    func New(n int) *Ring  // 初始化环
    func (r *Ring) Do(f func(interface{}))  // 循环环进行操作
    func (r *Ring) Len() int // 环长度
    func (r *Ring) Link(s *Ring) *Ring // 连接两个环
    func (r *Ring) Move(n int) *Ring // 指针从当前元素开始向后移动或者向前(n 可以为负数)
    func (r *Ring) Next() *Ring // 当前元素的下一个元素
    func (r *Ring) Prev() *Ring // 当前元素的上一个元素
    func (r *Ring) Unlink(n int) *Ring // 从当前元素开始，删除 n 个元素
```

12.4 本章小结

本章介绍了常用的几个 I/O 标准包的原理、使用以及注意事项，还针对常见的集中数据结构与算法的特点进行了介绍。在后续的章节里，还会接触文件操作与数据处理的相关标准库。

仅供非商业用途或交流学习使用



第13章 文件处理

在前面的学习中，几乎每个例子都有标准库的身影，标准库为大部分程序的编写提供了必要的基础功能，本章内容根据功能分类介绍文件处理相关的标准库，主要包括“文件操作”与“数据处理”相关的包。

Web 开发中对于文本处理是非常重要的一部分，往往需要对输出或者输入的内容进行处理，这里的文本包括字符串、数字、JSON、XML 等。Go 语言作为一门高性能的语言，对这些文本的处理都有官方的标准库来支持。使用过程中你会发现 Go 语言标准库的一些设计相当巧妙，而且对于使用者来说也很方便就能处理这些文本。本章将通过四节内容，让用户对 Go 语言处理文本有一个很好的认识。

文件操作最早是在函数章节中使用 defer 语句的时候遇到的，在后续的章节中也多次遇到文件操作的知识，第一节将整合之前遇到相关的知识点，介绍常见的文件操作。

13.1 文件操作

XML 是目前很多标准接口的交互语言，很多时候和 Java 编写的一些 Web Server 进行交互都是基于 XML 标准进行的，本章后面将介绍如何处理 XML 文本。

XML 有时候还是显得有些复杂，现在很多互联网企业对外的 API 是采用 JSON 格式的，这种格式虽然描述简单，但是能很好地表达意思，第三节将讲述如何处理这样的 JSON 格式数据。

Web 开发中一个很重要的部分就是 MVC 分离，在 Go 语言的 Web 开发中有一个专门的包来支持——template，最后一节将详细讲解如何使用模板来输出内容。

仅供非商业用途或交流学习使用



在任何计算机设备中，文件是都是必需的对象，而在编程中，文件的操作一直是程序员经常遇到的问题，例如生成文件目录、文件编辑等操作，本节把 Go 语言中的这些操作进行详细总结并介绍如何使用。

大多数文件操作的函数都是在 os 包里面的，下面列举了几个目录操作：

- func Mkdir(name string, perm FileMode) error 创建名称为 name 的目录，权限设置是 perm，例如 0777。
- func MkdirAll(path string, perm FileMode) error 根据 path 创建多级子目录，例如 zuolan/test1/test2。
- func Remove(name string) error 删除名称为 name 的目录，当目录下有文件或者其他目录时会出错
- func RemoveAll(path string) error 根据 path 删除多级子目录，如果 path 是单个名称，那么该目录下的子目录全部删除。

```
// 示例代码 13-1
package main

import (
	"fmt"
	"os"
)

func main() {
	os.Mkdir("goDir", 0777)
	os.MkdirAll("goDir/test1/test2", 0777)
	err := os.Remove("goDir")
	if err != nil {
		fmt.Println(err)
	}
	os.RemoveAll("goDir")
}
```

13.1.1 创建文件与查看状态

1. 新建文件

新建文件可以通过如下两个方法实现。

1）func Create(name string) (file *File, err Error)

根据提供的文件名创建新的文件，返回一个文件对象，默认权限是 0666 的文件，返

仅供非商业用途或交流学习使用

回的文件对象是可读写的。

2）func NewFile(fd uintptr, name string) *File

根据文件描述符创建相应的文件，返回一个文件对象。

2. 新建文件夹

新建文件夹的方法如下：

func MkdirAll(path string, perm FileMode) error

path 表示目录名及子目录，perm 表示目录权限，error 的值为 nil 表示创建成功无异常，如果目录已经存在不会执行操作。

3. 文件/文件夹状态

查看文件/文件夹状态的方法如下：

func Stat(name string) (FileInfo, error)

如果没有错误将返回描述命名文件的 FileInfo。

下面代码创建一个文件夹 test，在 test 里面新建一个文件夹 test2，然后在 test2 里面新建一个空白文件 demo，创建之后使用 Stat 函数获取相应文件或者文件夹的信息：

```
// 示例代码 13-2
package main

import (
	"fmt"
	"log"
	"os"
)

var (
	newFile  *os.File
	fileInfo os.FileInfo
	err      error
	path     = "test/test2/"
	fileName = "demo"
	filePath = path + fileName
)

func main() {
	// 创建文件夹
	err = os.MkdirAll(path, 0777)
	if err != nil {
```

仅供非商业用途或交流学习使用


```
            fmt.Printf("%s", err)
        } else {
            fmt.Println("成功创建目录。")
        }
        // 创建空白文件
        newFile, err = os.Create(filePath)
        if err != nil {
            log.Fatal(err)
        }
        fmt.Println(newFile)
        newFile.Close()

        // 查看文件的信息，如果文件不存在，则返回错误
        fileInfo, err = os.Stat(filePath)
        if err != nil && os.IsNotExist(err) {
            log.Fatal("文件不存在。")
        } else {
            log.Fatal(err)
        }
        fmt.Println("文件名称：", fileInfo.Name())
        fmt.Println("文件大小：", fileInfo.Size())
        fmt.Println("文件权限：", fileInfo.Mode())
        fmt.Println("最后修改时间：", fileInfo.ModTime())
        fmt.Println("是否是文件夹：", fileInfo.IsDir())
        fmt.Printf("系统接口类型：%T\n", fileInfo.Sys())
        fmt.Printf("系统信息：%+v\n\n", fileInfo.Sys())
}
/*
返回：
成功创建目录。
&{0xc42007a140}
文件名称：demo
文件大小：0
文件权限：-rw-rw-r--
最后修改时间：2017-10-11 15:32:48.035285203 +0800 HKT
是否是文件夹：false
系统接口类型：*syscall.Stat_t
系统信息：&{Dev:44 Ino:4924395 Nlink:1 Mode:33204 Uid:1000 Gid:1000 X__pad0:0 Rdev:
0 Size:0 Blksize:4096 Blocks:0 Atim:{Sec:1507706812 Nsec:923156583} Mtim:{Sec:1507
707168 Nsec:35285203} Ctim:{Sec:1507707168 Nsec:35285203} X__unused:[0 0 0]}
*/
```

在创建文件夹或者文件时，权限是一次性指定的，后续若要修改文件权限，需要使用其他函数（稍后介绍）。

判断文件是否存在可以使用函数 os.IsNotExist(err)，这个函数可以通过传入的 err 参数判断文件是否存在并返回一个布尔值（见上面例子）。

仅供非商业用途或交流学习使用

13.1.2　重命名与移动

在 Go 语言中，重命名和移动文件/文件夹都是相同的原理，使用函数：

```
func Rename(oldpath, newpath string) error
```

示例代码：

```
// 示例代码 13-3
package main

import (
    "log"
    "os"
)

func main() {
    originalPath := "test.txt"
    newPath := "test2.txt"
    err := os.Rename(originalPath, newPath)
    if err != nil {
        log.Fatal(err)
    }
}
```

13.1.3　打开与关闭

直接看下面的例子：

```
// 示例代码 13-4
package main

import (
      "log"
      "os"
)

var (
      file     *os.File
      fileInfo os.FileInfo
      err      error
      dirPath  = "test/test2/"
      fileName = "demo"
      filePath = dirPath + fileName
)

func main() {
      // 简单地以只读的方式打开（读写的例子稍后再说）。
```

仅供非商业用途或交流学习使用


```
	file, err := os.Open(dirPath)
	if err != nil {
		log.Fatal(err)
	}
	// 打印文件内容
	buf := make([]byte, 1024)
	for {
		n, _ := file.Read(buf)
		if 0 == n {
			break
		}
		os.Stdout.Write(buf[:n])
	}
	file.Close()

	/* OpenFile 提供更多的选项，第一个参数是文件路径，第二个参数是打开时的属性，第三个
参数是打开时的文件权限模式*/
	file, err = os.OpenFile(dirPath, os.O_APPEND, 0666)
	if err != nil {
		log.Fatal(err)
	}
	file.Close()
}
/*
返回文件内容。
*/
```

通过如下两个方法来打开文件：

- func Open(name string) (file *File, err Error)：该方法打开一个名称为 name 的文件，但用的是只读方式，内部实现其实是调用了 OpenFile()函数。
- func OpenFile(name string, flag int, perm uint32) (file *File, err Error)：打开名称为 name 的文件，flag 是打开的方式，包括只读、读写等，perm 是权限。

OpenFile()函数的第二个参数的属性可以单独使用，也可以组合使用。组合使用时可以使用 OR 操作符，例如：

```
os.O_CREATE|os.O_APPEND
os.O_CREATE|os.O_TRUNC|os.O_WRONLY
// os.O_RDONLY // 只读
// os.O_WRONLY // 只写
// os.O_RDWR   // 读写
// os.O_APPEND // 往文件中添加（Append）
// os.O_CREATE // 如果文件不存在则先创建
// os.O_TRUNC  // 文件打开时裁剪文件
// os.O_EXCL   // 和 O_CREATE 一起使用，文件不能存在
// os.O_SYNC   // 以同步 I/O 的方式打开
```

仅供非商业用途或交流学习使用

13.1.4　删除与截断

创建文件自然就有删除，用法如下：

```
// 示例代码 13-5
package main

import (
    "log"
    "os"
)

func main() {
    err := os.Remove("test.txt")
    if err != nil {
        log.Fatal(err)
    }
}
```

删除文件的函数非常简单，除了删除函数，Go 语言还提供了一个截断函数用于处理文件大小：

```
// 示例代码 13-6
package main

import (
    "log"
    "os"
)

func main() {
    err := os.Truncate("test.txt", 100)
    if err != nil {
        log.Fatal(err)
    }
}
```

上面例子中裁剪一个文件到 100 字节，如果文件本来就少于 100 字节，则文件中原始内容得以保留，剩余的字节以 null 字节填充。

如果文件本来超过 100 字节，则超过的字节会被抛弃，这样我们总是得到精确的 100 字节的文件。而如果传入 0 则会清空文件。

13.1.5　读写文件

读写文件中最常见操作有：复制文件、编辑、跳转、替换等，本节内容将介绍这些最

仅供非商业用途或交流学习使用

基础的读写操作，它们可以相互配合完成复杂的文件操作。

1. 复制文件

复制文件示例代码如下：

```
// 示例代码 13-7
package main

import (
	"io"
	"log"
	"os"
)

var (
	newFile  *os.File
	fileInfo os.FileInfo
	err      error
	path     = "test/test2/"
	fileName = "demo"
	filePath = path + fileName
)

func main() {
	// 打开原始文件
	originalFile, err := os.Open(filePath)
	if err != nil {
		log.Fatal(err)
	}
	defer originalFile.Close()
	// 创建新的文件作为目标文件
	newFile, err := os.Create(filePath + "_copy")
	if err != nil {
		log.Fatal(err)
	}
	defer newFile.Close()
	// 从源文件中复制字节到目标文件
	bytesWritten, err := io.Copy(newFile, originalFile)
	if err != nil {
		log.Fatal(err)
	}
	log.Printf("文件已复制，大小 %d bytes。", bytesWritten)
	// 将文件内容 flush 到硬盘中
	err = newFile.Sync()
	if err != nil {
		log.Fatal(err)
	}
```

仅供非商业用途或交流学习使用


```
}
/*
返回:
2017/10/11 19:45:58 文件已复制，大小 12 字节。
*/
```

复制操作有两点需要注意的地方：第一点是 Create 函数执行之后需要 Close()函数关闭回收资源；第二点是调用 io 包中的复制函数之后文件内容并没有真正保存在文件中，使用 Sync()函数同步之后才真正保存到硬盘中。

2. 跳转函数

跳转到文件指定位置是复杂的文件操作基础，在 Go 语言中，跳转函数为 Seek()，示例代码如下：

```
// 示例代码 13-8
package main

import (
	"fmt"
	"log"
	"os"
)

var (
	file     *os.File
	err      error
	dirPath  = "test/test2/"
	fileName = "demo"
	filePath = dirPath + fileName
)

func main() {
	file, _ := os.Open(filePath)
	defer file.Close()

	// 偏离位置，可以是正数也可以是负数
	var offset int64 = 5

	// 用来计算 offset 的初始位置
	// 0 = 文件开始位置
	// 1 = 当前位置
	// 2 = 文件结尾处
	whence := 0

	newPosition, err := file.Seek(offset, whence)
	if err != nil {
```

```
            log.Fatal(err)
        }
        fmt.Println("移动到位置 5: ", newPosition)
        // 从当前位置回退两字节
        newPosition, err = file.Seek(-2, 1)
        if err != nil {
            log.Fatal(err)
        }
        fmt.Println("从当前位置退回两字节: ", newPosition)
        // 使用下面的方式得到当前的位置
        currentPosition, err := file.Seek(0, 1)
        fmt.Println("当前位置: ", currentPosition)
        // 转到文件开始处
        newPosition, err = file.Seek(0, 0)
        if err != nil {
            log.Fatal(err)
        }
        fmt.Println("转到文件开始位置（0,0）: ", newPosition)
}
/*
返回:
移动到位置 5: 5
从当前位置退回两字节: 3
当前位置: 3
转到文件开始位置（0,0）: 0
*/
```

Seek()函数的特点类似于鼠标光标的定位，指定位置之后可以执行复制、剪切、粘贴等操作。接下来看如何在文件指定位置写入内容。

3. 写入函数

写入函数有下面几种：

- func (file *File) Write(b []byte) (n int, err Error) 写入 byte 类型的信息到文件。
- func (file *File) WriteAt(b []byte, off int64) (n int, err Error) 在指定位置开始写入 byte 类型的信息。
- func (file *File) WriteString(s string) (ret int, err Error) 写入 string 信息到文件。

写入函数示例代码如下：

```
// 示例代码 13-9
package main

import (
    "log"
    "os"
```

```
)

var (
	file     *os.File
	err      error
	dirPath  = "test/test2/"
	fileName = "demo"
	filePath = dirPath + fileName
)

func main() {
	// 可写方式打开文件
	file, err := os.OpenFile(filePath, os.O_WRONLY|os.O_TRUNC|os.O_CREATE, 0666)
	if err != nil {
		log.Fatal(err)
	}
	defer file.Close()
	// 将字节写入文件中
	file.Write([]byte("写入字节。\r\n"))
	// 将字符串写入文件中
	file.WriteString("写入字符串。\r\n")
	// 打印文件内容
	file, err = os.Open(filePath)
	if err != nil {
		log.Fatal(err)
	}
	buf := make([]byte, 1024)
	for {
		n, _ := file.Read(buf)
		if 0 == n {
			break
		}
		os.Stdout.Write(buf[:n])
	}
	file.Close()
}
/*
返回:
写入字符串。
写入字节。
*/
```

13.1.6　权限控制

查看文件（夹）权限示例代码如下：

```
// 示例代码 13-10
package main
```

```
import (
	"log"
	"os"
)

var (
	file     *os.File
	err      error
	dirPath  = "test/test2/"
	fileName = "demo"
	filePath = dirPath + fileName
)

func main() {
	// 测试写权限
	file, err := os.OpenFile(filePath, os.O_WRONLY, 0666)
	if err != nil && os.IsPermission(err) {
		log.Fatal("错误：没有写入权限。")
	} else if os.IsNotExist(err) {
		log.Fatal("错误：文件不存在。")
	} else {
		log.Fatal(err)
	}
	file.Close()
	// 测试读权限
	file, err = os.OpenFile(filePath, os.O_RDONLY, 0666)
	if err != nil && os.IsPermission(err) {
		log.Fatal("错误：没有读取权限。")
	} else if os.IsNotExist(err) {
		log.Fatal("错误：文件不存在。")
	} else {
		log.Fatal(err)
	}
	file.Close()
}
/*
根据情况返回不同内容。
*/
```

上面例子中如果没有写权限则返回 error，注意文件不存在也会返回 error，所以需要检查 error 的信息来确定到底是哪个错误导致的。

Go 语言自然也可以改变文件的权限，除此之外还可以改变文件的拥有者以及时间戳（最后修改时间）。修改文件权限和所有者类似于 Linux 下的 chmod 和 chown 命令，直接调用相应函数即可，示例代码如下：

```
// 示例代码 13-11
package main

import (
	"fmt"
	"log"
	"os"
	"time"
)

var (
	file     *os.File
	fileInfo os.FileInfo
	err      error
	dirPath  = "test/test2/"
	fileName = "demo"
	filePath = dirPath + fileName
)

func main() {
	// 使用 Linux 风格改变文件权限
	err := os.Chmod(filePath, 0777)
	if err != nil {
		log.Println(err)
	}
	// 改变文件所有者
	err = os.Chown(filePath, os.Getuid(), os.Getgid())
	if err != nil {
		log.Println(err)
	}

	// 查看文件信息
	fileInfo, err = os.Stat(filePath)
	if err != nil {
		if os.IsNotExist(err) {
			log.Fatal("文件不存在。")
		}
		log.Fatal(err)
	}
	fmt.Println("最后修改时间：", fileInfo.ModTime())
	// 改变时间戳
	twoDaysFromNow := time.Now().Add(48 * time.Hour)
	lastAccessTime := twoDaysFromNow
	lastModifyTime := twoDaysFromNow
	err = os.Chtimes(filePath, lastAccessTime, lastModifyTime)
	if err != nil {
		log.Println(err)
	}
```

```
}
/*
返回：
最后修改时间：2017-10-13 17:26:40.023074758 +0800 HKT
*/
```

执行上面程序两次之后，可以看到文件时间戳前后的变化。

13.1.7 文件链接

在 Linux 系统中肯定会经常遇到硬链接或者软链接之类的文件，对于一个普通文件，它实际上指向了硬盘的一个 inode 地址。硬链接会创建一个新的指针并且指向同一个地方，硬链接会保持与原文件双向同步，其中一个文件改动另一个文件也会改动，但只有所有的链接被删除后文件才会被删除（即移动和重命名都不会影响硬链接）。硬链接只在相同的文件系统中才能工作。

软连接和硬链接不一样，它不直接指向硬盘中的相同的地方，而是通过名字引用其他文件，它们可以指向不同的文件系统中的不同文件。Windows 操作系统不支持软链接。示例代码如下：

```
// 示例代码 13-12
package main

import (
	"fmt"
	"log"
	"os"
)

var (
	newFile  *os.File
	fileInfo os.FileInfo
	err      error
	path     = "test/test2/"
	fileName = "demo"
	filePath = path + fileName
)

func main() {
	// 创建一个硬链接
	// 创建后同一个文件内容会有两个文件名，改变一个文件的内容会影响另一个
	// 删除和重命名不会影响另一个
	hardLink := filePath + "hl"
	err := os.Link(filePath, hardLink)
```

```
        if err != nil {
              log.Fatal(err)
        }
        fmt.Println("创建硬链接")

        // 创建一个软链接
        softLink := filePath + "sl"
        err = os.Symlink(fileName, softLink)
        if err != nil {
              log.Fatal(err)
        }
        fmt.Println("创建软链接")

        /*Lstat 返回一个文件的信息，但是当文件是一个软链接时，它返回软链接的信息，而不是
          引用的文件的信息*/
        // Symlink 在 Windows 中不工作
        fileInfo, err := os.Lstat(softLink)
        if err != nil {
              log.Fatal(err)
        }
        fmt.Printf("链接信息：%+v", fileInfo)

        // 改变软链接的拥有者不会影响原始文件
        err = os.Lchown(softLink, os.Getuid(), os.Getgid())
        if err != nil {
              log.Fatal(err)
        }
  }
  /*
  返回：
  创建硬链接
  创建软链接
  链接信息：&{name:demosl size:4 mode:134218239 modTime:{wall:91593231 ext:6364331645
  7 loc:0x543c40} sys:{Dev:44 Ino:4927661 Nlink:1 Mode:41471 Uid:1000 Gid:1000 X__pa
  d0:0 Rdev:0 Size:4 Blksize:4096 Blocks:8 Atim:{Sec:1507719657 Nsec:91593231} Mtim:
  {Sec:1507719657 Nsec:91593231} Ctim:{Sec:1507719657 Nsec:91593231} X__unused:[0 0
  0]}}
  */
```

从 os.Lstat()的函数名中可以看出这是一个针对软链接的函数，用于查看软链接自己的属性，使用 os.Stat()函数会获取软链接指向的原文件信息。

此外需要注意软链接和硬链接实现的异同，从上面这两个函数的第一个参数来看，虽然都是 oldname，但实际例子中传递给函数的并不是同一个函数，硬链接是 filePath，而软链接是 fileName——因为硬链接是从项目根目录开始创建硬链接的，而软链接是根据目标文件的相对位置创建软链接的。

13.2 XML 处理

XML 作为一种数据交换和信息传递的格式已经十分普及，而随着 Web 服务日益广泛应用，现在 XML 在日常的开发工作中也扮演了愈发重要的角色。这一节将介绍 Go 语言标准包中的 XML 相关处理的包。

这一节不会涉及 XML 规范相关的内容（如需了解相关知识请参考其他文献），而是介绍如何用 Go 语言来编解码 XML 文件相关的知识。

假如你是一名运维人员，为所管理的所有服务器生成了如下内容的 XML 的配置文件：

```
<?xml version="1.0" encoding="utf-8"?>
<servers version="1">
      <server>
            <serverName>Local_Web</serverName>
            <serverIP>172.0.0.1</serverIP>
      </server>
      <server>
            <serverName>Local_DB</serverName>
            <serverIP>172.0.0.2</serverIP>
      </server>
</servers>
```

这个 XML 文档描述了两个服务器的信息，包含了服务器名和服务器的 IP 信息，接下来的 Go 语言例子以此 XML 描述的信息进行操作。

13.2.1 解析 XML

现在可以通过 xml 包的 Unmarshal 函数来解析 XML 文件：

```
func Unmarshal(data []byte, v interface{}) error
```

data 接收的是 XML 数据流，v 是需要输出的结构，定义为 interface，也就是可以把 XML 转换为任意格式。这里主要介绍 struct 的转换，因为 struct 和 XML 都有类似树结构的特征。示例如下：

```
// 示例代码 13-13
package main

import (
      "encoding/xml"
      "fmt"
      "io/ioutil"
      "os"
```

```
)

type Recurlyservers struct {
      XMLName     xml.Name `xml:"servers"`
      Version     string   `xml:"version,attr"`
      Svs         []server `xml:"server"`
      Description string   `xml:",innerxml"`
}

type server struct {
      XMLName    xml.Name `xml:"server"`
      ServerName string   `xml:"serverName"`
      ServerIP   string   `xml:"serverIP"`
}

func main() {
      file, err := os.Open("servers.xml") // 用于读取访问
      if err != nil {
            fmt.Printf("error: %v", err)
            return
      }
      defer file.Close()
      data, err := ioutil.ReadAll(file)
      if err != nil {
            fmt.Printf("error: %v", err)
            return
      }
      v := Recurlyservers{}
      err = xml.Unmarshal(data, &v)
      if err != nil {
            fmt.Printf("error: %v", err)
            return
      }

      fmt.Println(v)
}
```

XML 本质上是一种树形的数据格式，可以定义与之匹配的 Go 语言的 struct 类型，然后通过 xml.Unmarshal 来将 XML 中的数据解析成对应的 struct 对象。如上例子输出以下数据：

```
{{ servers} 1 [{{ server} Local_Web 172.0.0.1} {{ server} Local_DB 172.0.0.2}]
<server>
      <serverName>Local_Web</serverName>
      <serverIP>172.0.0.1</serverIP>
</server>
<server>
      <serverName>Local_DB</serverName>
```

```
        <serverIP>172.0.0.2</serverIP>
    </server>
    }
```

上面的例子中，将 XML 文件解析成对应的 struct 对象是通过 xml.Unmarshal 来完成的，这个过程是如何实现的呢？可以看到 struct 定义后面多了一些类似于 xml:"serverName" 这样的内容，这是 struct 的一个特性，被称为 struct tag，是用来辅助反射的。我们来看一下 Unmarshal 的定义：

```
func Unmarshal(data []byte, v interface{}) error
```

可以看到函数定义了两个参数，第一个是 XML 数据流，第二个是存储的对应类型，目前支持 struct、slice 和 string，xml 包内部采用了反射进行数据的映射，所以 v 里面的字段必须是导出的。Unmarshal 解析的时候 XML 元素和字段是怎样对应起来的呢？这是有一个优先级读取流程的，首先会读取 struct tag，如果没有，那么就会读取对应字段名。必须注意的一点是，解析的时候 tag、字段名、XML 元素都是大小写敏感的，所以字段必须一一对应。

Go 语言的反射机制，可以利用这些 tag 信息将来自 XML 文件中的数据反射成对应的 struct 对象，关于反射如何利用 struct tag 的更多内容请参阅 reflect 中的相关内容。

解析 XML 到 struct 的时候遵循如下的规则：

- 如果 struct 的一个字段是 string 或者[]byte 类型且它的 tag 含有",innerxml"，Unmarshal 会将此字段所对应的元素内所有内嵌的原始 XML 累加到此字段上，如上面例子中的 Description 定义。最后的输出是：

```
<server>
      <serverName>Local_Web</serverName>
      <serverIP>172.0.0.1</serverIP>
</server>
<server>
      <serverName>Local_DB</serverName>
      <serverIP>172.0.0.2</serverIP>
</server>
```

- 如果 struct 中有一个叫作 XMLName，且类型为 xml.Name 的字段，那么在解析的时候就会保存这个 element 的名字到该字段，如上面例子中的 servers。
- 如果某个 struct 字段的 tag 定义中含有 XML 结构中 element 的名称，那么解析的时候就会把相应的 element 值赋给该字段，如上面例子中的 servername 和 serverip 定义。

- 如果某个 struct 字段的 tag 定义了中含有",attr"，那么解析的时候就会将该结构所对应的 element 的与字段同名的属性的值赋给该字段，如上 version 定义。
- 如果某个 struct 字段的 tag 定义形如"a>b>c"，则解析的时候，会将 XML 结构 a 下面的 b 下面的 c 元素的值赋给该字段。
- 如果某个 struct 字段的 tag 定义了"-"，那么不会为该字段解析匹配任何 XML 数据。
- 如果 struct 字段后面的 tag 定义了",any"，如果它的子元素在不满足其他规则的时候就会匹配到这个字段。
- 如果某个 XML 元素包含一条或者多条注释，那么这些注释将被累加到第一个 tag 含有",comments"的字段上，这个字段的类型可能是[]byte 或 string，如果没有这样的字段存在，那么注释将会被抛弃。

上面详细讲述了如何定义 struct 的 tag。只要设置对了 tag，那么 XML 解析就如上面示例般简单，tag 和 XML 的 element 是一一对应的关系，还可以通过 slice 来表示多个同级元素。

为了正确解析，Go 语言的 xml 包要求 struct 定义中的所有字段必须是可导出的（即首字母大写）。

13.2.2　生成 XML

xml 包中提供了 Marshal 和 MarshalIndent 两个函数用于输出 XML，这两个函数的主要区别是第二个函数会增加前缀和缩进，函数的定义如下所示：

```
func Marshal(v interface{}) ([]byte, error)
func MarshalIndent(v interface{}, prefix, indent string) ([]byte, error)
```

两个函数的第一个参数用来生成 XML 的结构定义类型数据，都是返回生成的 XML 数据流。下面来看看如何输出如上的 XML：

```
// 示例代码 13-14
package main

import (
    "encoding/xml"
    "fmt"
    "os"
)

type Servers struct {
```

```
        XMLName xml.Name `xml:"servers"`
        Version string   `xml:"version,attr"`
        Svs     []server `xml:"server"`
}
type server struct {
        ServerName string `xml:"serverName"`
        ServerIP   string `xml:"serverIP"`
}
func main() {
        v := &Servers{Version: "1"}
        v.Svs = append(v.Svs, server{"Local_Web", "172.0.0.1"})
        v.Svs = append(v.Svs, server{"Local_DB", "172.0.0.2"})
        output, err := xml.MarshalIndent(v, "  ", "    ")
        if err != nil {
                fmt.Printf("error: %v\n", err)
        }
        os.Stdout.Write([]byte(xml.Header))

        os.Stdout.Write(output)
}
```

上面的代码输出如下信息：

```
<?xml version="1.0" encoding="UTF-8"?>
<servers version="1">
<server>
        <serverName>Local_Web</serverName>
        <serverIP>172.0.0.1</serverIP>
</server>
<server>
        <serverName>Local_DB</serverName>
        <serverIP>172.0.0.2</serverIP>
</server>
</servers>
```

和之前定义的文件的格式一模一样，之所以会有 os.Stdout.Write([]byte(xml.Header)) 这句代码的出现，是因为 xml.MarshalIndent 或者 xml.Marshal 输出的信息都是不带 XML 头的，为了生成正确的 XML 文件，使用了 xml 包预定义的 Header 变量。

我们看到 Marshal 函数接收的参数 v 是 interface{}类型，即它可以接收任意类型的参数，xml 包会根据下面的规则来生成相应的 XML 文件：

- 如果 v 是 array 或者 slice，那么便输出每一个元素，类似 value。
- 如果 v 是指针，那么会输出 Marshal 指针指向的内容，如果指针为空，什么都不输出。

- 如果 v 是 interface，那么就处理 interface 所包含的数据。
- 如果 v 是其他数据类型，就会输出这个数据类型所拥有的字段信息。

生成的 XML 文件中的 element 的名字根据如下优先级从 struct 中获取：

- 如果 v 是 struct，XMLName 的 tag 中定义的名称。
- 类型为 xml.Name 的名叫 XMLName 的字段的值。
- 通过 struct 中字段的 tag 来获取。
- 通过 struct 的字段名来获取。
- marshall 的类型名称。

设置 struct 中字段的 tag 信息以控制最终 XML 文件的生成：

- XMLName 不会被输出。
- tag 中含有"-"的字段不会输出。
- tag 中含有"name,attr"，会以 name 作为属性名，字段值作为值输出为这个 XML 元素的属性，如上 version 字段所描述。
- tag 中含有",attr"，会以这个 struct 的字段名作为属性名输出为 XML 元素的属性，类似上一条，只是这个 name 默认是字段名。
- tag 中含有",chardata"，输出为 XML 的 character data 而非 element。
- tag 中含有",innerxml"，将会被原样输出，而不会进行常规的编码过程。
- tag 中含有",comment"，将被当作 XML 注释来输出，而不会进行常规的编码过程，字段值中不能含有"--"字符串。
- tag 中含有"omitempty"，如果该字段的值为空值，那么该字段就不会被输出到 XML，空值包括：false、0、nil 指针或 nil 接口，任何长度为 0 的 array、slice、map 或者 string。
- tag 中含有"a>b>c"，那么就会循环输出三个元素，a 包含 b，b 包含 c，例如如下代码：

```
FirstName string   `xml:"name>first"`
LastName  string   `xml:"name>last"`
```

```
<name>
      <first>Asta</first>
      <last>Xie</last>
</name>
```

除了 XML 文件，JSON 也是非常流行的格式。

13.3 JSON 处理

JSON（JavaScript Object Notation）是一种轻量级的数据交换语言，以文字为基础，具有自我描述性且易于阅读。尽管 JSON 是 JavaScript 的一个子集，但 JSON 是独立于语言的文本格式，并且采用了类似于 C 语言家族的一些习惯。JSON 与 XML 最大的不同在于 XML 是一个完整的标记语言，而 JSON 不是。JSON 由于比 XML 更小、更快、更易解析，以及浏览器的内建快速解析支持，使得其更适用于网络数据传输领域。目前我们看到很多的开放平台，基本上都是采用了 JSON 作为数据交互的接口。既然 JSON 在 Web 开发中如此重要，那么 Go 语言对 JSON 支持得怎样呢？Go 语言的标准库已经非常好地支持了 JSON，可以很容易地对 JSON 数据进行编解码的工作。

前一小节的运维的例子用 JSON 来表示，结果描述如下：

```
{"servers":[{"serverName":"Local_Web","serverIP":"172.0.0.1"},{"serverName":"Local
_DB","serverIP":"172.0.0.2"}]}
```

本节余下的内容将以此 JSON 数据为基础，来介绍 Go 语言的 json 包对 JSON 数据的编解码。

13.3.1 解析 JSON

解析 JSON 有两种方法，一种是解析到结构体，另一种是解析到接口，前者是在知晓被解析的 JSON 数据的结构的前提下采取的方案，如果我们不知道被解析的数据的格式，则应该采用解析到接口的方案。

现在先看第一种，假如有了上面例子中的 JSON 串，那么如何来解析这个 JSON 串呢？Go 的 json 包中有如下函数：

```
func Unmarshal(data []byte, v interface{}) error
```

通过这个函数就可以实现解析的目的，详细的解析例子请看如下代码：

```
// 示例代码 13-15
package main

import (
      "encoding/json"
      "fmt"
)

type Server struct {
```

```
        ServerName string
        ServerIP   string
}

type Serverslice struct {
        Servers []Server
}

func main() {
        var s Serverslice
        str := `{"servers":[{"serverName":"Local_Web","serverIP":"172.0.0.1"},{"serv
erName":"Local_DB","serverIP":"172.0.0.2"}]}`
        json.Unmarshal([]byte(str), &s)
        fmt.Println(s)
}
```

在上面的示例代码中，首先定义了与 JSON 数据对应的结构体，数组对应 slice，字段名对应 JSON 里面的 KEY。在解析的时候，如何将 JSON 数据与 struct 字段相匹配呢？例如 JSON 的 key 是 Foo，那么怎样找对应的字段呢？

（1）首先查找 tag 中含有 Foo 的可导出的 struct 字段（首字母大写）。

（2）其次查找字段名是 Foo 的导出字段。

（3）最后查找类似 FOO 或者 FoO 这样的除了首字母之外其他大小写不敏感的导出字段。

注意，能够被赋值的字段必须是可导出字段（即首字母大写）。同时 JSON 解析的时候只会解析能找得到的字段，找不到的字段会被忽略，这样的一个好处是：当接收到一个很大的 JSON 数据结构而你却只想获取其中的部分数据的时候，只需将想要的数据对应的字段名大写，即可轻松解决这个问题。但如果遇到未知的 JSON 格式，则应该采用下面的方案。

我们知道 interface{}可以用来存储任意数据类型的对象，这种数据结构正好用于存储解析的未知结构的 JSON 数据的结果。json 包中采用 map[string]interface{}和[]interface{}结构来存储任意的 JSON 对象和数组。Go 语言类型和 JSON 类型的对应关系是 bool 代表 JSONbooleans，float64 代表 JSON numbers，string 代表 JSON strings，nil 代表 JSON null。

现在假设有某 JSON 数据，在不知道它的结构的情况下，把它解析到 interface{}里面，这里使用了 Bitly 公司开源的 simplejson 包，在处理未知结构体的 JSON 时比官方包更加方便，详细例子如下所示：

```
js, err := NewJson([]byte(`{
      "test": {
            "array": [1, "2", 3],
            "int": 10,
            "float": 5.150,
            "bignum": 9223372036854775807,
            "string": "simplejson",
            "bool": true
      }
}`))

arr, _ := js.Get("test").Get("array").Array()
i, _ := js.Get("test").Get("int").Int()
ms := js.Get("test").Get("string").MustString()
```

详细内容请参考 GitHub 地址：https://github.com/bitly/go-simplejson。

13.3.2 生成 JSON

很多应用最后都是要输出 JSON 数据串的，那么如何来处理呢？JSON 包里面通过 Marshal 函数来处理，函数定义如下：

```
func Marshal(v interface{}) ([]byte, error)
```

假设还是需要生成前面的服务器列表信息，那么如何来处理呢？请看下面的例子：

```
// 示例代码 13-16
package main

import (
      "encoding/json"
      "fmt"
)

type Server struct {
      ServerName string
      ServerIP   string
}

type Serverslice struct {
      Servers []Server
}

func main() {
      var s Serverslice
      s.Servers = append(s.Servers, Server{ServerName: "Local_Web", ServerIP:
"172.0.0.1"})
      s.Servers = append(s.Servers, Server{ServerName: "Local_DB", ServerIP:
"172.0.0.2"})
```

```
    b, err := json.Marshal(s)
    if err != nil {
        fmt.Println("json err:", err)
    }
    fmt.Println(string(b))
}
```

输出如下内容：

```
{"Servers":[{"ServerName":"Local_Web","ServerIP":"172.0.0.1"},{"ServerName":"Local
_DB","ServerIP":"172.0.0.2"}]}
```

可以看到输出字段名的首字母都是大写的，如果想用小写的首字母，那么直接把结构体的字段名改成首字母小写是不行的。JSON 输出的时候必须注意，只有导出的字段才会被输出，如果修改字段名，那么就会发现什么都不会输出，所以必须通过 struct tag 定义来实现：

```
type Server struct {
    ServerName string `json:"serverName"`
    ServerIP   string `json:"serverIP"`
}
type Serverslice struct {
    Servers []Server `json:"servers"`
}
```

通过修改结构体定义，输出的 JSON 串就和最开始定义的 JSON 串保持一致了。

针对 JSON 的输出，在定义 struct tag 的时候需要注意如下几点：

- 字段的 tag 是"-"，那么这个字段不会输出到 JSON。
- tag 中带有自定义名称，那么这个自定义名称会出现在 JSON 的字段名中，例如上面例子中的 serverName。
- tag 中如果带有"omitempty"选项，那么如果该字段值为空，就不会输出到 JSON 串中。
- 如果字段类型是 bool、string、int、int64 等，而 tag 中带有",string"选项，那么这个字段在输出到 JSON 的时候会把该字段对应的值转换成 JSON 字符串。

举例如下：

```
type Server struct {
    // ID 不会导出到 JSON 中
    ID int `json:"-"`

    // ServerName2 的值会进行二次 JSON 编码
```

```
        ServerName  string `json:"serverName"`
        ServerName2 string `json:"serverName2,string"`

        // 如果 ServerIP 为空，则不输出到 JSON 串中
        ServerIP   string `json:"serverIP,omitempty"`
    }

    s := Server {
        ID:         3,
        ServerName:  `Go "1.0" `,
        ServerName2: `Go "1.0" `,
        ServerIP:   ``,
    }
    b, _ := json.Marshal(s)
    os.Stdout.Write(b)
    /*
    返回:
    {"serverName":"Go \"1.0\" ","serverName2":"\"Go \\\"1.0\\\" \""}
    */
```

Marshal 函数只有在转换成功的时候才会返回数据，在转换的过程中需要注意几点：

- JSON 对象只支持 string 作为 key，所以要编码一个 map，那么必须是 map[string]T 这种类型（T 是 Go 语言中任意的类型）。
- Channel、complex 和 function 是不能被编码成 JSON 的。
- 嵌套的数据是不能编码的，不然会让 JSON 编码进入死循环。
- 指针在编码的时候会输出指针指向的内容，而空指针会输出 null。

本节介绍了如何使用 Go 语言的 json 标准包来编解码 JSON 数据，同时也简要介绍了如何在一些情况下使用第三方包 go-simplejson 简化操作。

13.4 日志记录

Go 语言中提供了一个简易的 log 包，使用该包可以方便地实现日志记录的功能，这些日志都是基于 fmt 包的打印再结合 panic 之类的函数来进行一般的打印、抛出错误处理的。

目前 Go 语言标准包只是包含了简单的功能，如果想把应用日志保存到文件，然后又能够结合日志实现很多复杂的功能，可以使用第三方开发的日志系统 Logrus（https://github.com/sirupsen/logrus）或 Seelog（https://github.com/cihub/seelog），它们实现了强大的日志功能，可以结合自己项目选择。

13.4.1　Logrus

logrus 是用 Go 语言实现的一个日志系统，与标准库 log 完全兼容并且核心 API 很稳定，是 Go 语言目前最活跃的日志库。首先安装 logrus：

```
go get -u github.com/sirupsen/logrus
```

简单例子如下：

```
// 示例代码 13-17
package main

import (
      log "github.com/Sirupsen/logrus"
)

func main() {
      log.WithFields(log.Fields{
            "tool": "pen",
      }).Info("This is pen.")
}
```

下面使用 logrus 自定义日志处理：

```
// 示例代码 13-18
package main

import (
      "os"
      log "github.com/Sirupsen/logrus"
)

func init() {
      // 日志格式化为 JSON 而不是默认的 ASCII
      log.SetFormatter(&log.JSONFormatter{})

      // 输出 stdout 而不是默认的 stderr，也可以是一个文件
      log.SetOutput(os.Stdout)

      // 只记录严重或以上警告
      log.SetLevel(log.WarnLevel)
}

func main() {
      log.WithFields(log.Fields{
            "tool": "pen",
            "price": 10,
      }).Info("The pen price is 10 dollars.")
```

```
        // }).Warn("The pen price is 10 dollars.")
        // }).Fatal("The pen price is 10 dollars.")

        // 通过日志语句重用字段
        // logrus.Entry 返回自 WithFields()
        contextLogger := log.WithFields(log.Fields{
                "common": "这是一个字段",
                "other":  "其他你想记录的东西",
        })

        contextLogger.Info("此处会记录 common 和 other 字段。")
}
```

13.4.2 Seelog

Seelog 是用 Go 语言实现的一个日志系统，它提供了一些简单的函数来实现复杂的日志分配、过滤和格式化。它拥有 XML 的动态配置，可以不用重新编译程序而动态加载配置信息；支持热更新，能够动态改变配置而不需要重启应用。此外还支持多输出流，能够同时把日志输出到多种流中，例如文件流、网络流等；支持不同的日志输出，包括命令行输出、文件输出、缓存输出等，支持 log rotate 和 SMTP 邮件提醒。

上面只列举了部分特性，Seelog 是一个特别强大的日志处理系统，详细内容请参看官方文档。接下来将简要介绍如何在项目中使用它，首先安装 Seelog：

```
go get -u github.com/cihub/seelog
```

然后来看一个简单的例子：

```
package main

import log "github.com/cihub/seelog"

func main() {
    defer log.Flush()
    log.Info("你好，Seelog。")
}
```

编译后运行，如果出现了“你好，Seelog。”，说明 Seelog 日志系统已经成功安装并且可以正常运行了。使用 Seelog 自定义日志处理可以查看官方文档，此处限于篇幅不再展开。

13.5　压缩

13.5.1　打包与解包

打包文件可以调用 archive 包，例子如下：

```
// 示例代码 13-19
// 这个例子使用 zip，但标准库也支持 tar 格式
package main

import (
    "archive/zip"
    "log"
    "os"
)

func main() {
    // 创建一个打包文件
    outFile, err := os.Create("test.zip")
    if err != nil {
        log.Fatal(err)
    }
    defer outFile.Close()

    // 使用 zip 包的创建函数 zipWriter 用于写入文件
    zipWriter := zip.NewWriter(outFile)


    // 往打包文件中写文件
    /* 这里使用硬编码的内容，你可以遍历一个文件夹，把文件夹下的文件以及它们的内容写入这个
打包文件中 */
    var filesToArchive = []struct {
        Name, Body string
    } {
        {"test.txt", "String contents of file"},
        {"test2.txt", "\x61\x62\x63\n"},
    }

    // 下面将要打包的内容写入打包文件中，依次写入
    for _, file := range filesToArchive {
            fileWriter, err := zipWriter.Create(file.Name)
            if err != nil {
                    log.Fatal(err)
            }
            _, err = fileWriter.Write([]byte(file.Body))
            if err != nil {
                    log.Fatal(err)
```

```
            }
        }

        // 清理
        err = zipWriter.Close()
        if err != nil {
            log.Fatal(err)
        }
    }
```

解包同样使用 archive 包中的函数：

```
// 示例代码 13-20
package main

import (
    "archive/zip"
    "log"
    "io"
    "os"
    "path/filepath"
)

func main() {
    zipReader, err := zip.OpenReader("test.zip")
    if err != nil {
        log.Fatal(err)
    }
    defer zipReader.Close()

    // 遍历打包文件中的每一文件/文件夹
    for _, file := range zipReader.Reader.File {
        // 打包文件中的文件就像普通的一个文件对象一样
        zippedFile, err := file.Open()
        if err != nil {
            log.Fatal(err)
        }
        defer zippedFile.Close()

        // 指定提取的文件名
        // 你可以指定全路径名或者一个前缀，这样可以把它们放在不同的文件夹中
        // 这个例子使用打包文件中相同的文件名
        targetDir := "./"
        extractedFilePath := filepath.Join(
            targetDir,
            file.Name,
        )

        // 提取项目或者创建文件夹
```

```
        if file.FileInfo().IsDir() {
            // 创建文件夹并设置同样的权限
            log.Println("正在创建目录：", extractedFilePath)
            os.MkdirAll(extractedFilePath, file.Mode())
        } else {
            // 提取正常的文件
            log.Println("正在提取文件：", file.Name)

            outputFile, err := os.OpenFile(
                extractedFilePath,
                os.O_WRONLY|os.O_CREATE|os.O_TRUNC,
                file.Mode(),
            )
            if err != nil {
                log.Fatal(err)
            }
            defer outputFile.Close()

            // 通过 io.Copy 简洁地复制文件内容
            _, err = io.Copy(outputFile, zippedFile)
            if err != nil {
                log.Fatal(err)
            }
        }
    }
}
```

13.5.2　压缩与解压

压缩文件使用的是 compress 包中的函数，示例如下：

```
// 示例代码 13-21
// 这个例子中使用 gzip 压缩格式，标准库还支持 zlib、bz2、flate、lzw
package main

import (
    "os"
    "compress/gzip"
    "log"
)

func main() {
    outputFile, err := os.Create("test.txt.gz")
    if err != nil {
        log.Fatal(err)
    }

    gzipWriter := gzip.NewWriter(outputFile)
```

```
    defer gzipWriter.Close()

    // 当写入 gizp writer 数据时，它会依次压缩数据并写入底层的文件中
    // 不必关心它是如何压缩的，还是像普通的 writer 一样操作即可
    _, err = gzipWriter.Write([]byte("Gophers rule!\n"))
    if err != nil {
        log.Fatal(err)
    }

    log.Println("已经压缩数据并写入文件。")
}
```

解压文件的例子：

```
// 示例代码 13-22
// 这个例子使用 gzip 压缩格式，标准库还支持 zlib、bz2、flate、lzw
package main

import (
    "compress/gzip"
    "log"
    "io"
    "os"
)

func main() {
    // 打开一个 gzip 文件
    /* 文件是一个 reader，但是可以使用各种数据源，比如 Web 服务器返回的 gzipped 内容，
       它的内容不是一个文件，而是一个内存流 */
    gzipFile, err := os.Open("test.txt.gz")
    if err != nil {
        log.Fatal(err)
    }

    gzipReader, err := gzip.NewReader(gzipFile)
    if err != nil {
        log.Fatal(err)
    }
    defer gzipReader.Close()

    // 解压缩到一个 writer，它是一个 file writer
    outfileWriter, err := os.Create("unzipped.txt")
    if err != nil {
        log.Fatal(err)
    }
    defer outfileWriter.Close()

    // 复制内容
    _, err = io.Copy(outfileWriter, gzipReader)
```

```
    if err != nil {
        log.Fatal(err)
    }
}
```

13.6　本章小结

本章主要介绍了文件和数据操作的相关标准库，文件处理包括日志处理都是在后端开发中非常重要的一部分，本章根据实际情况介绍了大部分输入/输出场景，包括文本、字符串、数字、JSON、XML 等。Go 语言作为一门现代化语言，对这些文件的处理都提供了官方的标准库，对于使用者来说很方便，也容易上手。

第14章 安全与测试

安全编程和测试一直是开发过程中不可或缺的一部分，不过一般来说，安全问题不是编程语言需要讨论的问题，但是 Go 语言是一门专门面向网络的新时代语言，对网络的支持也已融入其设计中，因此网络数据安全及其相应的体系就成了必须探讨的话题。

另一方面，测试作为传统开发中的一个比较突出的环节，Go 语言也没有忽视。使用 Go 语言的测试框架，可以在开发的过程中就进行单元测试和基准测试。和 go build 命令类似，go test 命令可以用来执行写好的测试代码，需要做的就是遵守一些规则来写测试。而且，Go 语言可以将测试无缝地集成到代码工程和持续集成系统里。

14.1 安全

信息数据化和传输网络化对数据和数据传输的安全提出了要求。在这两个环节上，需要对数据进行加密，并使用安全的数据传输体系。本节将着重介绍常见的安全编程的概念以及 Go 语言安全方面的函数等的用法。

14.1.1 安全相关的基础概念

1. 数据加密

采用单密钥的加密算法，称为对称加密。整个系统由如下几部分构成：需要加密的明文、加密算法和密钥。在加密和解密中，使用的密钥只有一个。常见的单密钥加密算法有 DES、AES、RC4 等。

采用双密钥的加密算法，称为非对称加密。整个系统由如下几个部分构成：需要加密

的明文、加密算法、私钥和公钥。在该系统中，私钥和公钥都可以被用作加密或者解密，但是用私钥加密的明文，必须要用对应的公钥解密，用公钥加密的明文，必须用对应的私钥解密。常见的双密钥加密算法有 RSA 等。

在对称加密中，私钥不能暴露，否则在算法公开的情况下，数据等同于明文，而在非对称加密中，公钥是公开的，私钥是保密的。这样任何人都可以把自己的信息通过公钥和算法加密，然后发送给公钥的发布方，只有公钥发布方才能解开密文。

可以看到，在对称加密和非对称加密中，它们有一个共同的特点，即数据可以加密，也可以解密。实际上，还有一种加密需求，只需要加密，形成一个密文，而不需要解密，甚至极端地说，要求不可解密。这时候，可以使用哈希算法等。

哈希算法是一种从任意数据中创建固定长度摘要信息的办法。一般要求，对于不同的数据，要求产生的摘要信息也是唯一的。常见的哈希算法包括 MD5、SHA-1 等。

2．数字签名

数字签名，是指用于标记数字文件拥有者、创造者、分发者身份的字符串。数字签名拥有标记文件身份、分发的不可抵赖性等作用。

目前，常用的数字签名采用了非对称加密。例如，A 公司发布了一个可执行文件，称为 AProduct，A 在 AProduct 中加入了 A 公司的数字签名。A 公司的数字签名是用 A 公司的私钥加密了 AProduct 文件的哈希值，得到打过数字签名的 AProduct 后，可以查看数字签名。

这个过程实际上是用 A 公司的公钥解密了文件哈希值，从而可以验证两个问题：AProduct 是否由 A 公司发布，以及 AProduct 是否被篡改。

3．数字证书

假设登录某银行的网站，这时候网站会提示下载数字证书，否则将无法正常使用网银等功能。在首次使用 U 盾的时候，初始化过程即是向 U 盾中下载数字证书。那么，数字证书中包含什么呢？数字证书中包含了银行的公钥，有了公钥之后，网银就可以用公钥加密提供给银行的信息，这样只有银行才能用对应的私钥得到我们的信息，确保安全。

4．PKI 体系

PKI，全称公钥基础设施，是使用非对称加密理论，提供数字签名、加密、数字证书等服务的体系，一般包括权威认证机构（CA）、数字证书库、密钥备份及恢复系统、证书

作废系统、应用编程接口（API）等。

围绕 PKI 体系，建立了一些权威的、公益的机构。它们提供数字证书库、密钥备份及恢复系统、证书作废系统、应用接口等具体的服务。比如，企业的数字证书，需要向认证机构申请，以确保数字证书的安全。

14.1.2 通信安全

1. 哈希函数

Go 语言提供了 MD5、SHA-1 等几种哈希函数，下面的例子是一个简单的介绍：

```
// 示例代码 14-1
package main

import (
      "crypto/md5"
      "crypto/sha1"
      "fmt"
)

func main() {
      TestString := "Hi,Golang!"
      Md5Inst := md5.New()
      Md5Inst.Write([]byte(TestString))
      Result := Md5Inst.Sum([]byte(""))
      fmt.Printf("%x", Result)
      fmt.Printf("\n")
      Sha1Inst := sha1.New()
      Sha1Inst.Write([]byte(TestString))
      Result = Sha1Inst.Sum([]byte(""))
      fmt.Printf("%x", Result)
}
/*
返回:
465982b657a9ca520e2e233ef2c6b094
01c858d49ef76be4ef548962e72586c8f222ae9d
*/
```

再举一个例子，对文件内容计算 SHA1：

```
// 示例代码 14-2
package main

import (
      "crypto/md5"
      "crypto/sha1"
```

仅供非商业用途或交流学习使用


```
        "fmt"
        "io"
        "os"
)

func main() {
        fileName := "chapter14/14-1.go"
        infile, inerr := os.Open(fileName)
        if inerr == nil {
                md5h := md5.New()
                io.Copy(md5h, infile)
                fmt.Printf("文件“%s”的 md5 值为：%x", fileName, md5h.Sum([]byte("")))
                fmt.Printf("\n")
                sha1h := sha1.New()
                io.Copy(sha1h, infile)
                fmt.Printf("文件“%s”的 sha1 值为：%x", fileName, sha1h.Sum([]byte("")))
        } else {
                fmt.Println(inerr)
                os.Exit(1)
        }
}
/*
返回：
文件“chapter14/14-1.go”的 md5 值为：0bcea8f3a3d5f1f424d90d4ce5acb9eb
文件“chapter14/14-1.go”的 sha1 值为：da39a3ee5e6b4b0d3255bfef95601890afd80709
*/
```

2. 加密通信流程

一般的 HTTPS 基于 SSL（Secure Sockets Layer）协议。SSL 是网景公司开发的位于 TCP 与 HTTP 之间的透明安全协议，通过 SSL 可以把 HTTP 包数据以非对称加密的形式往返于浏览器和站点之间，从而避免被第三方非法获取。

目前，伴随着电子商务的兴起，HTTPS 获得了广泛的应用。由 IETF（Internet Engineering Task Force）实现的 TLS（Transport Layer Security）是建立于 SSL v3.0 之上的兼容协议，它们的主要区别在于所支持的加密算法。

当用户在浏览器中输入一个以 https 开头的网址时，便开启了浏览器与被访问站点之间的加密通信。

下面以一个用户访问 https://baidu.com 为例，给读者展现 SSL/TLS 的工作方式。

- 在浏览器中输入 HTTPS 协议的网址。
- 服务器向浏览器返回证书。
- 浏览器检查该证书的合法性。

仅供非商业用途或交流学习使用

- 浏览器使用证书中的公钥加密一个随机对称密钥，并将加密后的密钥和使用密钥加密后的请求 URL 一起发送到服务器。
- 服务器用私钥解密随机对称密钥，并用获取的密钥解密加密的请求 URL。
- 服务器把用户请求的网页用密钥加密，并返回给用户。
- 用户浏览器用密钥解密服务器发来的网页数据，并将其显示出来。

上述过程都是依赖于 SSL/TLS 层实现的。在实际开发中，SSL/TLS 的实现和工作原理比较复杂，但基本流程与上面的过程一致。

SSL 协议由两层组成，上层协议包括 SSL 握手协议、更改密码规格协议、警报协议，下层协议包括 SSL 记录协议。

SSL 握手协议建立在 SSL 记录协议之上，在实际的数据传输开始前，用于在客户与服务器之间进行握手。握手是一个协商过程，这个协议使得客户和服务器能够互相鉴别身份，协商加密算法。在任何数据传输之前，必须先进行握手。在握手完成之后，才能进行 SSL 记录协议，它的主要功能是为高层协议提供数据封装、压缩、添加 MAC、加密等支持。

3. 支持 HTTPS 的 Web 服务器

Go 语言目前实现了 TLS 协议的部分功能，已经可以提供最基础的安全层服务。下面来看如何实现支持 TLS 的 Web 服务器。如下示例示范了如何使用 http.ListenAndServerTLS 实现一个支持 HTTPS 的 Web 服务器：

```
// 示例代码 14-3
package main

import (
      "fmt"
      "net/http"
)

const SERVER_PORT = 8080
const SERVER_DOMAIN = "localhost"
const RESPONSE_TEMPLATE = "Hello World"

func rootHandler(w http.ResponseWriter, req *http.Request) {
      w.Header().Set("Content-Type", "text/html")
      w.Header().Set("Content-Length", fmt.Sprint(len(RESPONSE_TEMPLATE)))
      w.Write([]byte(RESPONSE_TEMPLATE))
}
```

仅供非商业用途或交流学习使用


```
func main() {
    http.HandleFunc(fmt.Sprintf("%s:%d/", SERVER_DOMAIN, SERVER_PORT), rootHandl
er)
    http.ListenAndServeTLS(fmt.Sprintf(":%d", SERVER_PORT), "some.crt", "some.ke
y", nil)
}
```

这段代码实际上是无法运行的，因为并没有提供相应的证书文件，程序只会直接停止运行。如果一切准备好，运行该服务器程序后，在浏览器中访问 localhost:8080 可以看到一句“Hello World”。

可以看到代码中使用了 http.ListenAndServerTLS()这个方法，这表明它是执行在 TLS 层上的 HTTP 协议。相应地，非 TLS 协议只需要把该方法替换为 http.ListenAndServe (fmt.Sprintf(":%d", SERVER_PORT), nil)即可。

4. 支持 HTTPS 的文件服务器

利用 Go 语言标准库中提供的完备封装，也可以很容易地实现一个支持 HTTPS 的文件服务器，如下面代码所示：

```
// 示例代码 14-4
package main

import (
    "net/http"
)

func main() {
    h := http.FileServer(http.Dir("."))
    http.ListenAndServeTLS(":8001", "rui.crt", "rui.key", h)
}
```

这是一个极简的例子，只是单纯介绍了 FileServer 函数的存在，这个函数的参数只有一个就是根目录路径，其他任务全部由 ListenAndServe*等函数完成。

本节主要介绍了网络安全应用领域的相关知识点，以及 Go 语言对网络安全应用的全面支持，同时还提供了具有一定实用价值的示例，让读者可以更加具体地理解相关的知识，并能基于这些示例快速写出实用的程序。Go 语言标准库的网络和加解密等相关的包在设计上都做了一定程度的抽象，以大幅提高易用性，提高开发效率。

因为本书的重点在于介绍 Go 语言相关知识，所以对于安全相关的知识就不做非常深入的阐述了，如果读者对安全编程有兴趣，可自行阅读网络安全的书籍。

仅供非商业用途或交流学习使用

14.2 测试

在计算机编程中，单元测试（Unit Testing）又称为模块测试，是针对程序模块（软件设计的最小单位）来进行正确性检验的测试工作。程序单元是应用的最小可测试部件，在过程化编程中，一个单元就是单个程序，包括函数、过程等；对于面向对象编程，最小单元就是方法，包括基类（超类）、抽象类或者派生类（子类）中的方法。

测试的目的自然是确认代码是否正常工作，例如测试代码是否可以成功地向数据库中插入一条记录，这种测试叫作“正向路径”测试，就是在正常执行的情况下，保证代码不产生错误的测试。

另外一种情况是测试代码是否会产生预期的错误，例如程序对数据库进行查询时没有找到任何结果，或者对数据库做了无效的更新，那么应该返回一个可以控制的错误，而不是导致程序崩溃，这种测试即为“负向路径”的测试场景，保证代码不仅会产生错误，而且是预期的错误。

总之，不管如何调用或者执行代码，所写的代码行为都是可预期的测试才算通过。

在 Go 语言里有几种方法可用于单元测试，基础测试（basic test）是指只使用一组参数和结果来测试一段代码；表组测试（table test）是指使用多组参数和结果测试一段代码。此外，也可以使用一些方法来模仿（mock）测试代码需要使用到的外部资源，如数据库或者网络服务器。例如当外部资源不可用的时候，模拟这些资源的行为可以使测试正常进行。

最后，在构建自己的网络服务时，有几种方法可以在不运行服务的情况下，调用服务的功能进行测试。

14.2.1 单元测试

testing 是 Go 语言的一个 package，它提供自动化测试功能。通过 go test 命令能够自动执行如下形式的函数：

```
func TestXxx(*testing.T)
```

其中 Xxx 可以是任何字母、数字、字符串，但是 Xxx 的第一个字母不能是小写字母。在这些函数中，使用 Error、Fail 或相关方法来返回失败信号。

要编写一个新的测试模块，需要创建一个名称以_test.go 结尾的文件，该文件包含 TestXxx 函数，最后将该文件放在与被测试的包相同的包目录中。该文件将被排除在正常

仅供非商业用途或交流学习使用

的程序包之外，但在运行 go test 命令时将被调用。运行 go help test 和 go help testflag 可以了解更加详细的信息。

1. 第一个单元测试

要测试的代码如下，下面的 Age 函数中，如果输入的参数值小于 0 则返回 0，如果输入大于 0 则返回相应数值：

```
// 示例代码 14-5
package main

import "fmt"

func main() {
	fmt.Println(Age(-7))
}

// Age 测试
func Age(n int) int {
	if n > 0 {
		return n
	}
	n = 0
	return n
}
```

测试代码如下所示，现在测试的是如果输入小于 0 的数字，程序是否会返回相应的数字 0：

```
// 示例代码 14-6
package main

import "testing"

// TestFib 测试
func TestFib(t *testing.T) {
	var (
		input    = -100
		expected = 0
	)
	actual := Age(input)
	if actual != expected {
		t.Errorf("Fib(%d) = %d, 预期为 %d", input, actual, expected)
	}
}
```

执行测试命令：

仅供非商业用途或交流学习使用


```
$ go test .
ok      chapter13/basic_test 0.002s
```

这就是基础测试，下面来看表组测试，可以提供多组数据的测试方式。

2. 表组测试

测试讲究覆盖率，按上面的方式，当要覆盖更多情况时，显然通过修改代码的方式很笨拙。这时可以采用表组测试的方式写测试代码，标准库中有很多测试是使用这种方式写的。

例如，以下程序的作用是判断一个数字是否是素数：

```
// 示例代码 14-7
package main

import "fmt"

func main() {
      fmt.Println(isPrime(25))
}

// 大于 1 的自然数中，除了 1 和它本身以外不再有其他因数的数称为质数
func isPrime(value int) bool {
      if value <= 3 {
            return value >= 2
      }
      if value%2 == 0 || value%3 == 0 {
            return false
      }
      for i := 5; i*i <= value; i += 6 {
            if value%i == 0 || value%(i+2) == 0 {
                  return false
            }
      }
      return true
}
```

基于以上程序写测试，可以像下面这样写：

```
// 示例代码 14-8
package main

import "testing"

// TestPrime 测试
func TestPrime(t *testing.T) {
      var primeTests = []struct {
```

仅供非商业用途或交流学习使用


```
        input    int  // 输入
        expected bool // 期望结果
    }{
        {1, false},
        {2, true},
        {3, true},
        {4, false},
        {5, true},
        {6, false},
        {7, false},
    }

    for _, tt := range primeTests {
        actual := isPrime(tt.input)
        if actual != tt.expected {
            t.Errorf("%d %v %v", tt.input, actual, tt.expected)
        }
    }
}
```

上面测试中最后一个测试用例是错误的，所以执行测试时会返回测试不通过的结果，当然错误的原因并不是程序的问题，而是测试用例的错误。

```
$ go test .
--- FAIL: TestPrime (0.00s)
    /.../chapter13/basic_test/table_test.go:23: 7 true false
FAIL
exit status 1
FAIL  github.com/.../chapter13/table_test    0.002s
Error: Tests failed.
```

因为测试中使用的是 t.Errorf，其中某个情况测试失败，并不会中止测试，其他测试用例会继续执行下去。在单元测试中，传递给测试函数的参数是*testing.T 类型，它用于管理测试状态并支持格式化测试日志（测试日志会在执行测试的过程中不断累积，并在测试完成时输出到标准输出上）。

在一次测试中，测试函数执行结束返回，或者测试函数调用 FailNow、Fatal、Fatalf、SkipNow、Skip 和 Skipf 中的任意一个时，这次测试即宣告结束。与 Parallel 方法一样，以上提到的这些方法只能在运行测试函数的 goroutine 中调用。而其他打印的方法，比如 Log 以及 Error 的变种，则可以在多个 goroutine 中同时调用。

下面总结了测试的几个方法的含义，当某个测试用例测试失败时这些方法的后续动作分别如下。

- Fail：记录失败信息，然后继续执行后续用例。

仅供非商业用途或交流学习使用

- FailNow：记录失败信息，所有测试中断。
- SkipNow：不会记录失败的用例信息，然后中止测试。
- Skip：记录失败信息，中断后续测试。
- Skipf：相比前者多了一个格式化输出。
- Log：输出错误信息，在单元测试中，默认不输出成功的用例信息，不会中断后续测试。
- Logf：输出格式化的信息，不中断后续测试。
- Error：相当于 Log + Fail，不会中断后续测试。
- Errorf：相当于 Logf + Fail，同上。
- Fatal：相当于 Log + FailNow，会中断后续测试。
- Fatalf：相当于 Logf + FailNow，同上。

在默认情况下，单元测试成功时，它们打印的信息不会输出，可以通过加上-v 选项，输出这些信息。但对于基准测试，它们总是会被输出。

例如，调用*T 或*B 的 Skip 方法，跳过该测试或基准测试：

```
func TestPrime(t *testing.T) {
    if testing.Short() { // 快速测试的标记
        t.Skip("skipping test in short mode.")
    }
    ...
}
```

3. 模拟测试

单元测试的原则，就是你所测试的函数方法不受所依赖环境的影响，比如网络访问等。但有时候运行单元测试的时候需要联网，而由于开发环境限制，不能联网，此时就需要模拟网络访问来完成测试了。

针对模拟网络访问，标准库提供了一个 httptest 包，可以模拟 HTTP 的网络调用，下面举个例子了解如何使用。

首先创建一个处理 HTTP 请求的函数，并注册路由：

```
package common

import (
      "net/http"
      "encoding/json"
)
```

仅供非商业用途或交流学习使用


```
func Routes(){
      http.HandleFunc("/sendjson",SendJSON)
}
func SendJSON(rw http.ResponseWriter,r *http.Request){
      u := struct {
            Name string
      }{
            Name:"张三",
      }
      rw.Header().Set("Content-Type","application/json")
      rw.WriteHeader(http.StatusOK)
      json.NewEncoder(rw).Encode(u)
}
```

非常简单，这里是一个/sendjsonAPI，当访问这个 API 时，会返回一个 JSON 字符串。现在对这个 API 服务进行测试，但是又不能时时刻刻都启动着服务，所以这里就用到了外部终端对 API 的网络访问请求：

```
func init()  {
      common.Routes()
}

func TestSendJSON(t *testing.T){
      req,err:=http.NewRequest(http.MethodGet,"/sendjson",nil)
      if err!=nil {
            t.Fatal("创建 Request 失败")
      }

      rw:=httptest.NewRecorder()
      http.DefaultServeMux.ServeHTTP(rw,req)

      log.Println("code:",rw.Code)

      log.Println("body:",rw.Body.String())
}
```

运行这个单元测试，就可以看到程序自动访问/sendjsonAPI 的结果，并且没有启动任何 HTTP 服务就达到了目的。这里主要利用 httptest.NewRecorder()创建一个 http.ResponseWriter，模拟了真实服务器端的响应，这种响应是通过调用 http.DefaultServeMux. ServeHTTP 方法触发的。

还有一个模拟调用的方式，是真的在测试机上模拟一个服务器，然后进行调用测试：

```
func mockServer() *httptest.Server {
      //API 调用处理函数
      sendJson := func(rw http.ResponseWriter, r *http.Request) {
            u := struct {
```

```
                Name string
        }{
                Name: "张三",
        }

        rw.Header().Set("Content-Type", "application/json")
        rw.WriteHeader(http.StatusOK)
        json.NewEncoder(rw).Encode(u)
    }
    //适配器转换
    return httptest.NewServer(http.HandlerFunc(sendJson))
}

func TestSendJSON(t *testing.T) {
    //创建一个模拟的服务器
    server := mockServer()
    defer server.Close()
    //Get 请求发往模拟服务器的地址
    resq, err := http.Get(server.URL)
    if err != nil {
        t.Fatal("创建 Get 失败")
    }
    defer resq.Body.Close()

    log.Println("code:", resq.StatusCode)
    json, err := ioutil.ReadAll(resq.Body)
    if err != nil {
        log.Fatal(err)
    }
    log.Printf("body:%s\n", json)
}
```

模拟服务器的创建使用的是 httptest.NewServer 函数，它接收一个 http.Handler 处理 API 请求的接口。代码示例中使用了 Hander 的适配器模式，http.HandlerFunc 是一个函数类型，实现了 http.Handler 接口，注意这里是强制类型转换，不是函数的调用。

这个创建的模拟服务器，监听的是本机 IP 127.0.0.1，端口是随机的。接着发送 Get 请求的时候，不再发往/sendjson，而是模拟服务器的地址 server.URL，剩下的就和访问正常的 URL 一样了，打印出结果即可。

4. 测试覆盖率

尽可能模拟更多的场景来测试代码的不同情况，但是有时候的确也有忘记测试的代码，这时候就需要测试覆盖率作为参考了。

由单元测试的代码，触发运行的被测试代码的代码行数占所有代码行数的比例，被称

为测试覆盖率，代码覆盖率不一定完全精准，但是可以作为参考，可以有助于测量和预计的覆盖率之间的差距，go test 工具就提供了这样一个度量测试覆盖率的能力。

依旧使用前面素数判断的程序（注意把 7 对应的值修改为正确的，不然无法通过测试），现在使用 go test 工具运行单元测试，和前几次不一样的是，要显示测试覆盖率，所以要多加一个参数-coverprofile，完整的命令为：go test -v -coverprofile=c.out，-coverprofile 是指定生成的覆盖率文件，例子中是 c.out，这个文件稍后会用到。现在看终端输出，已经有了一个覆盖率：

```
$ go test -v -coverprofile=c.out
=== RUN   TestPrime
--- PASS: TestPrime (0.00s)
PASS
coverage: 66.7% of statements
ok      github.com/izuolan/example/chapter13/basic_test 0.003s
```

现在测试覆盖率为 66.7%，还没有到 100%，那么看看还有哪些代码没有被测试到。这就需要刚刚生成的测试覆盖率文件 c.out 生成测试覆盖率报告了。生成报告使用 Go 提供的工具 go tool cover -html=c.out -o=tag.html，即可生成一个名字为 tag.html 的 HTML 格式的测试覆盖率报告（如图 14-1 所示），这里有详细的信息告诉我们哪一行代码测试到了，哪一行代码没有测试到。

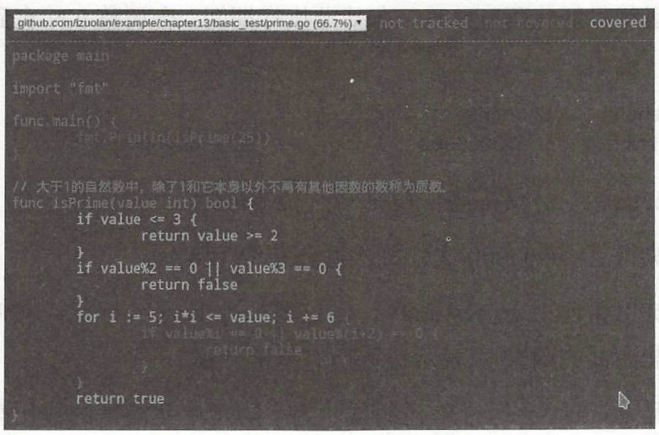


图 14-1　显示为测试的语句

可以看到，标记为绿色的代码行已经被测试了，标记为红色的还没有测试到，现在根据没有测试到的代码逻辑，完善单元测试代码即可（例如增加一个测试实例，输入为 25 即可 100%覆盖 isPrime 函数）。

14.2.2 基准测试

基准测试（benchmarking）是一种测量和评估软件性能指标的活动。在某个时候通过基准测试建立一个已知的性能水平（称为基准线），当系统的软硬件环境发生变化之后再进行一次基准测试以确定那些变化对性能的影响，这是基准测试最常见的用途。其他用途包括测定某种负载水平下的性能极限、管理系统或环境的变化、发现可能导致性能问题的条件等。

在_test.go 结尾的测试文件中，如下形式的函数（基准测试的函数必须以 Benchmark 开头，必须是可导出的）：

```
func BenchmarkXxx(*testing.B)
```

被认为是基准测试，通过 go test 命令，加上-bench 选项来执行。多个基准测试按照顺序运行。

基准测试函数样例如下所示：

```
func BenchmarkHello(b *testing.B) {
    for i := 0; i < b.N; i++ {
        fmt.Sprintf("hello")
    }
}
```

基准函数会运行目标代码 b.N 次。在基准执行期间，会调整 b.N 直到基准测试函数持续足够长的时间，输出：

```
$ go test -bench=. -run=none
BenchmarkSprintf-4      10000000               148 ns/op
PASS
ok     github.com/..../14/bench_test       2.249s
```

意味着循环执行了 10000000 次，每次循环花费 148 ns。

运行基准测试也要使用 go test 命令，不过要加上-bench=标记，它接受一个表达式作为参数，匹配基准测试的函数，.表示运行所有基准测试。

因为默认情况 go test 会运行单元测试，为了防止单元测试的输出影响查看基准测试的结果，可以使用-run=匹配一个不存在的单元测试方法，过滤掉单元测试的输出，这里使用 none，因为基本上不会创建这个名字的单元测试方法。

下面着重解释一下输出的结果，函数后面的-4 表示运行时对应的 GOMAXPROCS 的值；接着的 10000000 表示运行 for 循环的次数，也就是调用被测试代码的次数，最后的

148 ns/op 表示每次需要花费 148ns。

以上测试时间默认是 1s，也就是 1s 的时间调用一千万次，每次调用花费 148ns。如果想让测试运行的时间更长，可以通过-benchtime 指定，比如 3s：

```
$ go test -bench=. -benchtime=3s -run=none
```

当然，在一般情况下，加长测试时间，只会导致测试的次数变多，但是最终的性能结果并没有太大变化。建议测试时间不要超过 3s，当然，具体情况具体分析。

如果在运行前基准测试需要一些耗时的配置，则可以先重置定时器（例如避免 for 循环之前的初始化代码的干扰）：

```
func BenchmarkBigLen(b *testing.B) {
    ... ... // 初始化代码
    b.ResetTimer() //
    ... ... // 通常是一个 for 循环
}
```

如果基准测试需要在并行设置中测试性能，则可以使用 RunParallel 辅助函数，这样的基准测试一般与 go test -cpu 标志一起使用：

```
func BenchmarkTemplateParallel(b *testing.B) {
    templ := template.Must(template.New("test").Parse("Hello, {{.}}!"))
    b.RunParallel(func(pb *testing.PB) {
        // 每个 goroutine 有属于自己的 bytes.Buffer
        var buf bytes.Buffer
        for pb.Next() {
            // 所有 goroutine 一起，循环一共执行 b.N 次（N 指默认的 CPU 核心数）
            buf.Reset()
            templ.Execute(&buf, "World")
        }
    })
}
```

基准测试是一种测试代码性能的方法。想要测试解决同一问题的不同方案的性能，以及查看哪种解决方案的性能更好，基准测试就会很有用。基准测试也可以用来识别某段代码的 CPU 或者内存效率问题，而这段代码的效率可能会严重影响整个应用程序的性能。许多开发人员会用基准测试来测试不同的并发模式，或者用基准测试来辅助配置工作池的数量，以保证能最大化系统的吞吐量。

1. 性能对比

上面那个基准测试的例子，其实是一个 int 类型转为 string 类型的例子，标准库里还

有几种方法，看一下哪种性能更加高效，具体代码如下：

```
func BenchmarkSprintf(b *testing.B){
	num:=10
	b.ResetTimer()
	for i:=0;i<b.N;i++{
		fmt.Sprintf("%d",num)
	}
}
func BenchmarkFormat(b *testing.B){
	num:=int64(10)
	b.ResetTimer()
	for i:=0;i<b.N;i++{
		strconv.FormatInt(num,10)
	}
}
func BenchmarkItoa(b *testing.B){
	num:=10
	b.ResetTimer()
	for i:=0;i<b.N;i++{
		strconv.Itoa(num)
	}
}
```

在第一次调用时，b.N 的值为 1。需要注意的是，一定要将所有要进行基准测试的代码都放到循环里，并且循环要使用 b.N 的值，否则测试的结果是不可靠的。

执行基准测试：

```
$ go test -bench=. -run=none
goos: linux
goarch: amd64
pkg: github.com/izuolan/example/chapter13/bench_test
BenchmarkSprintf-4      10000000               155 ns/op
BenchmarkFormat-4       300000000                6.99 ns/op
BenchmarkItoa-4         200000000                9.79 ns/op
PASS
ok      github.com/izuolan/example/chapter13/bench_test 7.243s
```

从结果上看 strconv.FormatInt 函数是最快的，其次是 strconv.Itoa，而 fmt.Sprintf 最慢，前两个函数性能相对于最后一个超过了 3 倍。

为了进一步分析三个函数快慢的根源，可以通过-benchmem 分析内存的使用情况。-benchmem 可以提供每次操作分配内存的次数，以及每次操作分配的字节（B）数。

```
$ go test -bench=. -benchmem -run=none
goos: linux
goarch: amd64
```

```
pkg: github.com/izuolan/example/chapter13/bench_test
BenchmarkSprintf-4      10000000               150 ns/op              16 B/op
 2 allocs/op
BenchmarkFormat-4       300000000               5.01 ns/op            2 B/op
 1 allocs/op
BenchmarkItoa-4         200000000               7.00 ns/op            2 B/op
 1 allocs/op
PASS
ok      github.com/izuolan/example/chapter13/bench_test 5.811s
```

这次输出的结果会多出两组新的数值：一组数值的单位是 B/op，另一组的单位是 allocs/op，其中单位为 allocs/op 的值表示每次操作从堆上分配内存的次数。可以看到 Sprintf 函数每次操作都会从堆上分配两个值，而另外两个函数每次操作只需要分配一个值。单位为 B/op 的值表示每次操作分配的 B 数，你可以看到 Sprintf 函数两次分配总共消耗了 16B 的内存，而另外两个函数操作消耗 2B 内存。从这个数据就知道它为什么这么慢了，内存分配与占用都太高。

在运行单元测试和基准测试时，还有很多选项可以用。建议读者查看一遍所有选项（下一节会介绍一部分），以便在编写自己的包和工程时，充分利用测试框架。Go 社区希望包的作者在正式发布包的时候提供足够的测试。

在代码开发中，对于要求性能的地方，编写基准测试非常重要，这有助于开发出性能更好的代码。不过性能、可用性、复用性等也要有一个相对的取舍，不能为了追求性能而过度优化。

2. pprof

在之前的例子中，只能查看函数的执行时间，如果想进一步分析函数的具体执行状况可以配合其他选项使用。上面曾经使用到一些基准测试的选项，其中常用的测试选项如下。

- -bench regexp：regexp 可以是任何正则表达式，表示需要运行的基准测试函数。一般可以使用`-bench .`来执行当前目录下所有的基准测试。
- -benchmen：在输出内容中包含基准测试的内存分配统计信息。
- -benchtime t：t 表示执行单个测试函数的累积耗时上限，默认是 1s。
- -cpuprofile out：输出 cpu profile 到指定的路径，可以使用 pprof 查看。
- -memprofile out：输出内存 profile 到指定路径，可以使用 pprof 查看。

执行一个基准测试时，可以指定相关的选项，例如以下这样：

```
$ go test -bench . --benchmem -cpuprofile cpu.prof
goos: linux
```

```
goarch: amd64
pkg: github.com/izuolan/example/chapter13/bench_test
BenchmarkSprintf-4      10000000              176 ns/op              16 B/op
 2 allocs/op
BenchmarkFormat-4       300000000               4.90 ns/op            0 B/op
 0 allocs/op
BenchmarkItoa-4         200000000               6.95 ns/op            0 B/op
 0 allocs/op
PASS
ok      github.com/izuolan/example/chapter13/bench_test 6.213s
$ ls
bench_test.test  cpu.prof  str_test.go
```

生成两个文件是无法直接查看的，需要使用工具去解析。

这就需要本节的主角 pprof 登场了，这是一个 Go 语言提供的性能分析工具，可以分析 cpu profile、memory profile、heap profile、block profile 等信息。

在上面的基准测试中，已经知道怎样生成 cpu.prof 文件了，然后可以利用 go tool pprof 工具来查看：

```
$ go tool pprof bench_test.test cpu.prof
File: bench_test.test
Type: cpu
Time: Nov 14, 2017 at 7:27pm (HKT)
Duration: 6.21s, Total samples = 6.05s (97.49%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof)
```

这时会进入一个可交互环境，输入 help 可以查看所有的交互命令，现在可以使用 top10 来查看测试过程中最耗 CPU 资源的函数：

```
(pprof) top10
Showing nodes accounting for 4840ms, 80.00% of 6050ms total
Dropped 39 nodes (cum <= 30.25ms)
Showing top 10 nodes out of 39
      flat  flat%   sum%        cum   cum%
    2160ms 35.70% 35.70%     2160ms 35.70%  strconv.FormatInt /.../go/src/strconv/
                                             itoa.go
     840ms 13.88% 49.59%     1760ms 29.09%  strconv.Itoa /.../go/src/strconv/itoa.go
     710ms 11.74% 61.32%     1950ms 32.23%  .../bench_test.BenchmarkFormat .../bench_
                                            test/str_test.go
     350ms  5.79% 67.11%     2110ms 34.88%  .../bench_test.BenchmarkItoa .../bench_
                                             test/str_test.go
     200ms  3.31% 70.41%      310ms  5.12%  runtime.mallocgc /.../go/src/runtime/
                                            malloc.go
     150ms  2.48% 72.89%      340ms  5.62%  fmt.(*fmt).fmt_integer /.../go/src/fmt/
                                            format.go
```

```
    110ms   1.82%  74.71%  320ms   5.29%   fmt.(*pp).free /.../go/src/fmt/print.go
    110ms   1.82%  76.53%  360ms   5.95%   runtime.slicebytetostring /.../go/src/
                                           runtime/string.go
    110ms   1.82%  78.35%  190ms   3.14%   sync.(*Pool).pin /.../go/src/sync/pool.go
    100ms   1.65%  80.00%  100ms   1.65%   runtime.memmove /.../go/src/runtime/
                                           memmove_amd64.s
```

现在解释每一列的含义。在默认情况下，Go 语言的运行时系统会以 100Hz 的频率对 CPU 使用情况进行取样。也就是说每 s 取样 100 次，即每 10ms 会取样一次（100Hz 既足够产生有用的数据，又不至于让系统产生停顿）。实际上，这里所说的对 CPU 使用情况的取样就是对当前的 goroutine 的堆栈上的程序计数器的取样，由此就可以从样本记录中分析出哪些代码是计算时间最长或者说最耗 CPU 资源的部分了。

- 第一列表示取样点落在该函数里的总数（不包括调用其他函数）。比如 fmt/print 这个函数，总共执行时间为 110ms，那么总的抽样点数为 11 次。
- 第二列表示落在该函数里取样点占总取样点的百分比，fmt/print 这个函数占总共的 1.82%。
- 第三列表示的是前几行加起来的执行时间占总共执行时间的多少，top10 命令的默认排序是按第一列（执行时间）排序。例如第三行的第三列表示的是：前三行函数的执行时间加起来占总共执行时间的 61.32%。
- 第四列表示取样点落在该函数里和它直接调用、间接调用的函数里的总数。比如 fmt/print 这个函数，总时间为 320ms，表示该函数的执行时间加上该函数调用的其他函数的执行时间共 320ms。
- 第五列表示第四列的时间占总时间的百分比。

从 top10 的输出里至少可以知道每个函数的执行时间占比以及每个函数的调用栈的执行时间的占比。如果某个函数自身的执行时间过长，那说明这个函数是否有逻辑错误，是否需要拆分。如果某个函数的调用栈的执行时间过长，是否是因为调用了过多的不需要的函数。

有了初步认识之后，可以使用图形化的界面来分析整个函数的执行过程。在这之前，需要安装 Graphviz，具体安装方法见软件官网，安装好之后在 pprof 交互模式下使用 web 命令查看。

top 和 web 等命令也可以指定函数名称，只查看关注的函数的执行情况：

```
(pprof) top BenchmarkFormat
Showing nodes accounting for 1950ms, 32.23% of 6050ms total
```

```
      flat  flat%   sum%        cum   cum%
   1240ms 20.50% 20.50%     1240ms 20.50%  strconv.FormatInt /.../go/src/strconv/itoa.go
    710ms 11.74% 32.23%     1950ms 32.23%  .../bench_test.BenchmarkFormat /.../bench_
                                            test/str_test.go
         0     0% 32.23%     1950ms 32.23%  testing.(*B).launch /.../go/src/testing/
                                            benchmark.go
         0     0% 32.23%     1950ms 32.23%  testing.(*B).runN /.../go/src/testing/
                                            benchmark.go
```

更进一步而言，可以通过 list 命令查看具体是哪一行在耗时，如图 14-2 所示。

```
(pprof) list BenchmarkFormat
Total: 6.05s
ROUTINE ======================== github.com/izuolan/example/chapter13/b
ample/chapter13/bench_test/str_test.go
     710ms      1.95s (flat, cum) 32.23% of Total
         .          .     26:func BenchmarkFormat(b *testing.B) {
         .          .     27:   number := int64(10)
         .          .     28:
         .          .     29:   b.ResetTimer()
         .          .     30:
     650ms      650ms     31:   for i := 0; i < b.N; i++ {
      60ms      1.30s     32:           strconv.FormatInt(number, 10)
         .          .     33:   }
         .          .     34:}
```

图 14-2　函数执行状况

这样就能知道函数的整体执行情况了，第一列表示该行的执行时间，第二列表示该行的总执行时间。

14.3　本章小结

通过本章内容的学习，了解了最基本的 Go 语言安全方面的概念，主要掌握了如何测试一个 Go 语言程序，Go 语言提供了必要的测试工具，例如 go test 用来运行测试（测试文件总是以_test.go 作为文件名的结尾）。表组测试是利用一个测试函数测试多组值的好办法，基准测试可以探查代码性能，从而优化 Go 语言程序。

第15章 内存管理

Go 语言是一门内置运行时的编程语言，这种类型的编程语言一般都会抛弃传统的内存分配方式，改为自主管理。这种方式可以完成类似预分配、内存池等操作，以避开系统调用带来的性能问题。当然，还有一个重要原因是为了更好地配合垃圾回收。

编程语言的内存管理一直以来都是非常重要的话题，而 Go 语言的内存管理基于 TCMalloc（Google 推出的内存分配器），使用了类似 TCMalloc 的算法进行内存分配，而且 Go 语言在实现的时候还做了很多优化，本章将通过源码来看一下 Go 语言的内存管理实现。

15.1 内存分配

Go 语言的内存分配基于 TCMalloc，TCMalloc 是 Google 开发的一个内存分配器，在不少项目中都有使用，常见的内存分配器还有 glibc 的 PTMalloc 和同样是 Google 的 JEMalloc。相比于 PTMalloc，TCMalloc 性能更好，特别适用于高并发场景。TCMalloc 具有现代化内存分配器的基本特征：对抗内存碎片、在多核处理器上可伸缩。据称，它的内存分配速度是 glibc2.3 中实现的 malloc 的数倍。

在了解 TCMalloc 之前，需要认识一些内存管理基本概念。

15.1.1 内存管理基本概念

对于以前尚未接触过内存管理的读者而言，理解一些基本概念有助于后续的学习。如果是之前对其他语言的内存管理有经验的读者，不妨温习一次。

1. 内存地址

在几十年前，还是汇编主宰着计算机世界，那个时候内存管理还不算什么大问题，程序有多少用多少，不够就换设备。

但是在那个年代，不是每一个开发人员都会精打细算一个程序需要多少内存，如果内存空间有限，而内存的需求是不确定的，那么就需要一些手段来管理程序的内存使用了。早期的内存管理一般只有三步：首先确定内存是不是够用，够用就从内存中取一点来运行程序，用完了就把内存放回“内存池”中，负责这个过程的就是内存分配程序了。程序的动态性越强，内存管理就越重要，内存分配程序也就更重要。

时至今日，内存管理也不过是这三个步骤的进化。进化后的内存管理有三种类型：第一种是全手工的内存分配，它们认为内存很重要，不能由程序自己管理；第二种是全自动分配，它们认为内存很重要，不能交给人来管理；第三种是处于两者之间的半自动的内存管理方案。

不管是哪一种，都有一个绕不过去的地方，那就是“内存在程序中是如何分配的”，换句话说“如何将内存分配给程序”。要理解这一点，首先需要理解操作系统在其中担任的角色，计算机上的每一个进程都认为自己可以访问所有的物理内存，但实际上由于同时运行着多个程序，所以每个进程不可能拥有全部内存，这个时候就需要一种能够“欺骗”进程的东西，这就是虚拟内存。

要实现虚拟内存，首先要想办法避免让进程直接接触到实际的物理地址，因此物理内存会被操作系统组织成一个数组，这个数组由 N 个连续的字节（B）大小的单元组成，每个字节（B）都有一个唯一的物理地址（PA）。现代处理器使用的是一种称为虚拟寻址（VA）的寻址形式，最小的寻址单位是字，虚拟地址映射物理地址是通过读取页表（page table）进行地址翻译完成的——页表存放在物理存储器中。内核把物理页作为内存管理的基本单位，以页（page）为单位来管理内存中的页表。

举一个不太合理的例子，一个程序目前正在访问地址为 233 的内存，但实际上虚拟内存系统不需要将这个程序存储在位置为 233 的物理内存中。实际上，它甚至可以不存储在物理内存中，如果物理内存已经满了，它甚至可能已经被转移到硬盘上了。由于这类地址不必反映内存所在的物理位置，所以它们（如 233 的地址）被称为虚拟内存。

操作系统维持着一个虚拟地址到物理地址的转换表，以便计算机硬件可以正确地响应地址请求。并且，如果地址在硬盘上而不是在内存中，那么操作系统将暂时停止该进程（对

于存储在硬盘的内存数据，Windows 中称为“虚拟内存”，Linux 中称为 Swap）。当空间有余或者优先度提高之后，再从硬盘上加载被进程请求的内存，然后再重新启动进程。这样，每个进程都获得了自己可以使用的地址空间，可以访问比物理上安装的内存更多的内存。

理论上，64 位系统中每一个进程最多可以访问 128GB 内存，但实际上即便有 128GB 内存，再加上硬盘虚拟内存（Swap），每个进程所使用的内存也肯定少于 128GB。因为当加载一个进程时，它会得到一个取决于称为系统中断点（system break）的特定地址，这是一个初始内存分配的值，该地址之后是未被映射的内存，未被映射的内存是“不存在”的，也就不能被进程使用（没有分配相应虚拟内存地址的物理内存）。因此，如果一个进程运行超出了它的初始分配内存，那么它必须请求操作系统映射进来更多的内存（映射是一种表示一一对应关系的数学术语，内存虚拟地址对应一个物理地址，就称为内存被映射）。

以 Linux 系统为例，有两个可映射到物理内存的基本系统调用，分别是 brk()和 mmap()，这两个函数的作用是一样的——用来增减进程的虚拟内存。不过 brk()更加简单，它只是单纯将系统中断点（system break）向前或者向后移动，使得进程可以获得更多内存或者释放内存。

而 Go 语言在内存分配上的底层函数是 mmap()，相比 brk()函数，mmap()更为灵活。首先，它可以映射任何位置的内存，而不局限于进程；其次，它不仅可以将虚拟地址映射到物理内存或者虚拟内存（Swap），还可以将虚拟地址映射到文件和文件位置（文件路径），这样可以在读写内存时直接对文件中的数据进行读写。

2. 内存中的栈和堆

先说明，此处的堆栈并非数据结构中的堆栈，而是指内存中的一个区域。

在早期的计算机中，内存分配是基于栈的，这样的好处是显而易见的，速度快、操作方便，可以有效避免内存碎片化（栈是连续的）。但是栈的问题也逐渐暴露，例如不利于管理大内存（16 位时代之后）、数据的生命周期难于控制（栈内的有效数据通常是连续存储的，所以 pop 时后申请的内存必须早于先申请的内存失效），所以栈不利于动态地管理并且有效地利用宝贵的内存资源，于是就有了堆。

要知道“内存分配”与“垃圾回收”都属于“动态存储分配”，“动态存储分配器”维护的虚拟存储区域称为堆，堆可以视为一组大小不同的块（chunk：连续的虚拟存储片，内存分配器和垃圾回收算法都依赖连续地址）的集合，并交由动态存储器维护。

但是堆的内存分配效率相比栈而言太低了，所以现在的编程规范一般会比较在意堆栈的区分。

回到话题上，有动态分配自然也有静态分配的概念。图 15-1 展现了内存中的存储的结构。

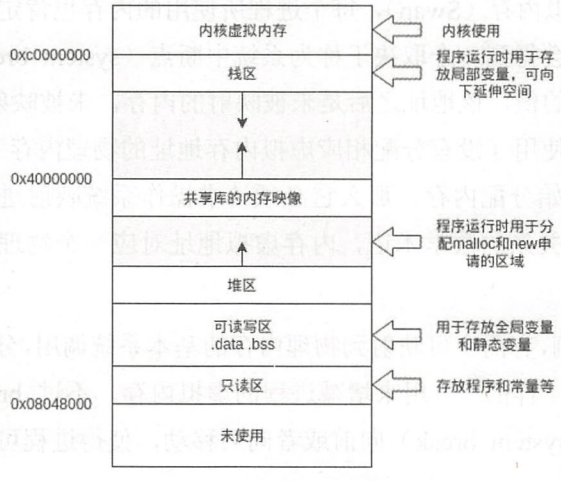


图 15-1　内存存储结构

内存中的栈区处于相对较高的地址，栈地址是向下增长的，而堆区的地址是向上增长的。栈区一般由操作系统自动分配释放，是一块连续的内存区域（数据结构是一个栈），堆区一般由程序员分配释放，或者由 OS 回收（数据结构上类似一个链表）。

静态分配是系统编译器完成的，比如局部变量的分配，也就是说一般的内存分配器会把局部变量分配到栈区。而堆区用于分配开发人员申请的内存空间。注意，静态变量、全局变量一般是分配到堆区的。只读区是分配常量和程序代码空间的。

既然堆栈性能差异大，那么应该如何分配对象到堆栈上呢？Go 语言的回答是：不用你操心。

这是 Go 语言的一个特性，编译器会在编译的时候自动判断对象应该分配到哪里，这当然很棒。但是如果不理解编译器的“自动分配”过程，很容易犯一些错误，毕竟编译器实际上也只是按照规矩办事，所以接下来看看 Go 语言编译器是根据何种规矩分配对象内存的。

15.1.2　逃逸分析

由于栈的性能要比堆好很多，分配内存的速度比堆快（小内存分配），所以在编码过程中，开发人员会比较在意一个变量或者对象被分配到哪里，是堆还是栈这对于程序的性能和安全都有较大影响。

于是就有了一个“逃逸分析”的概念，在编译程序优化理论中，逃逸分析是一种确定指针动态范围的方法，可以分析在程序的哪些地方可以访问到指针。当一个变量或对象在子程序中被分配内存时，一个指向变量或对象的指针可能逃逸到其他执行线程中，甚至去调用子程序。

例如一个子程序分配一个对象并且返回一个该对象的指针，该对象可能在程序中的任何一个地方被访问到——这样指针就成功“逃逸”了。如果指针存储在全局变量或者其他数据结构中，它们也可能发生逃逸，这种情况是当前程序中的指针逃逸。

以 Go 语言为例，注意下面两个函数分别创建了两个变量，这意味着两个函数的初始化都需要向 OS 申请内存，但是注意，它们申请到的内存位置是不一样的。

```
// 示例代码 15-1
package main
func toHeap() *int {
      var x int
      return &x
}
func toStack() int {
      x := new(int)
      *x = 1
      return *x
}
func main() {}
```

为了显示上面两个函数的区别，需要开启逃逸分析日志，在编译的时候加上-gcflags '-m'即可，不过为了不让编译器自动内连函数，一般会加-l 参数，所以最终编译参数为 -gcflags '-m -l'（-m 意思是打印编译器的自动优化策略，使用 go tool compile -help 查看所有可以被传入编译器的参数列表）。

```
$ go run -gcflags '-m -l' main.go
# command-line-arguments
./main.go:5:9: &x escapes to heap
./main.go:4:6: moved to heap: x
./main.go:8:10: toStack new(int) does not escape
```

仅供非商业用途或交流学习使用

上面的意思是 toHeap()中的 x 最后在堆上分配，而 toStack()中的 x 最后分配在栈上（尽管是通过 new 分配的）。

上面的 escape 判断就是一种“逃逸分析”，逃逸分析需要确定指针所有可以存储的地方，保证指针的生命周期只在当前进程或线程中。

那么同样是变量申请内存，为什么两个函数分配到的位置却不一样呢？

这是因为 Go 语言在一定程度上消除了堆和栈的区别，在编译的时候会自动进行逃逸分析，不逃逸的对象就放到栈上，可能逃逸的对象就放到堆上。

这样一看，Go 语言岂不是不能手动控制堆栈分配了？其实也不是，上文说的是“一定程度上消除了堆和栈的区别”，实际上开发人员还是可以手动控制内存分配到堆栈的，具体规则如下：

- Go 语言中的变量只要被引用就一直存活，存储在堆上还是栈上由内部实现决定，和具体的语法没有关系。
- 一般情况会将函数的局部变量分配到函数栈（stack frame）上。
- 如果变量在函数 return 之后还会被引用，编译器就会将变量分配到堆上。
- 如果一个局部变量非常大，那么它也会被分配到堆上而不是栈上。
- 如果一个变量被取地址（*和&操作），那么它很可能会被分配到堆上，最终分配结果还是要对这些变量做逃逸分析，如果函数 return 之后变量不再被引用，则将其分配到栈上。

下面以一个例子说明：

```
// 示例代码 15-2
package main

type S struct{}

func main() {
        var a S
        x := &a
        _ = toStack(x)
        _ = toHeap(a)
}

/*
这是一个“流水”函数，因为这个函数只是单纯地将 x 作为输入变量，然后将 x 变量返回，这个函数没
```

仅供非商业用途或交流学习使用


```
有引用 z。同时也没有引用 a，因为传入的是 a 的地址不是 a 的值，a 的生命周期从头到尾都在 main 函
数中，所以 a 变量没有逃逸的可能，就分配到了栈中
*/
func toStack(z *S) *S {
        return z
}

/*
记住一点，Go 语言默认是值传递的，所以此处输入的参数实际上是复制了 a 的值，传给了 z，然后函数
返回的是 z 的指针，上面说过取地址操作有可能被分配到堆上，而且 z 的地址作为返回值可能会被外部
访问，所以 z 有逃逸的可能，最终确定被分配到堆上
*/
func toHeap(z S) *S {
        return &z
}
```

执行编译，查看逃逸分析日志：

```
$ go run -gcflags '-m -l' main.go
# command-line-arguments
./main.go:12:18: leaking param: z to result ~r1 level=0
./main.go:17:9: &z escapes to heap
./main.go:16:16: moved to heap: z
./main.go:7:7: main &a does not escape
```

可以看到 Go 语言编译器在编译时就将堆栈的分配搞定了。但是如果 Go 语言编译器不知道两个变量的关系，会不会分配有误？

例如以下程序：

```
// 示例代码 15-3
package main

type S struct {
        M *int
}

func main() {
        var a int
        var b S
        ref(&a, &b)
}

func ref(x *int, y *S) {
        y.M = x
}
/*
```

仅供非商业用途或交流学习使用


```
# command-line-arguments
./ex.go:13:21: leaking param: x
./ex.go:13:21: ref y does not escape
./ex.go:10:6: &a escapes to heap
./ex.go:8:6: moved to heap: a
./ex.go:10:10: main &b does not escape
*/
```

在上述这个例子中，函数 ref 中的 x 赋值给 y 的成员（即 a 被外部引用了），所以 a 需要存在堆中，避免 a 丢失数据。接下来，b 自始至终都只在 main 中引用，即便在 ref 中也只是被赋值而已，不会在函数（子程序）外部被访问，没有逃逸的可能。

接下来尝试把上面的 y.M = x 语句反过来写：x = y.M，结果是全部不会逃逸，全都分配到栈上（没有外部引用）。

下一节将介绍 Go 语言内存分配器分配内存的原型——TCMalloc，说白了就是 Go 语言堆的内存分配原理，栈的内存分配比较简单，和 Go 语言自己的内存分配器一起讲解。

15.2 TCMalloc

TCMalloc 的全称是 Thread-Caching Malloc（线程缓存分配器），是 Google 公司推出的一个内存分配器，在不少项目中都有使用，Go 语言就是使用了类似的算法实现内存分配的，所以本节先来认识一下 TCMalloc。常见的内存分配器还有 glibc 的 PTMalloc 和同样是 Google 的 JEMalloc。相对于 PTMalloc，TCMalloc 特别对多线程做了优化，对于小对象的分配基本上不存在锁竞争，而大对象使用了细粒度、高效的自旋锁（spinlock），因此特别适用于高并发场景。

为了更好地理解 TCMalloc 工作过程，本节将从两个角度解释这个流程：一是从线程角度出发，根据请求分配内存的大小分为“小内存分配”和“大内存分配”；二是从 TCMalloc 角度出发，根据不同内存分配请求划分不同等级。如图 15-2 所示，可以看到“小内存分配”和 ThreadCache 对应，而“大内存分配”和 PageHeap 对应。

仅供非商业用途或交流学习使用

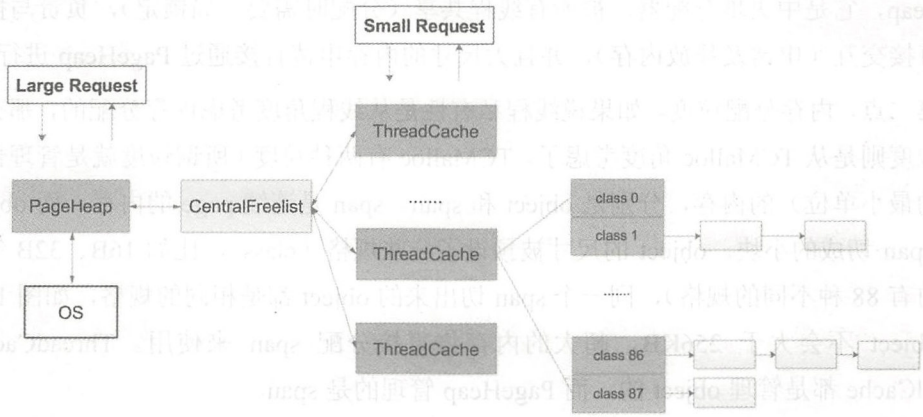


图 15-2　TCMalloc 结构图

15.2.1　整体结构

TCMalloc 内存管理体系分为三个层次：ThreadCache、CentralCache 和 PageHeap，如图 15-2 所示。分配内存和释放内存的时候都是在各个层次中逐级尝试的，整体的思路是当最小的 ThreadCache 层分配内存失败，则从下一层的 CentralCache 中分配一批补充上来，还不够就再从 PageHeap 那里要，再不够向操作系统要。在释放内存时，ThreadCache 层释放了过多的内存，则回收一批到 CentralCache 层，以此类推逐级回收到 PageHeap 并最终还给操作系统。现在我们了解的是一个整体结构，但至于为何要这样设计以及这样设计的好处在哪里，将是本节的重点。

这三个层次从前到后担负着不同的角色，在了解各角色担负的任务之前，需要了解 TCMalloc 的两个特点。

第一点，线程私有性。这是和 ThreadCache 有关的概念，顾名思义，ThreadCache 是每个线程都有一份的。在理想状况下，每个线程的内存申请都可以在自己的 ThreadCache 里面完成，线程之间不需要竞争，因此非常高效，这是 TCMalloc 的一大亮点。

然而现实是不可能达到上面的理想情况的，ThreadCache 不可能永远都能满足线程的需求，ThreadCache 中的内存不够的时候就需要 CentralCache 甚至 PageHeap 出马了。CentralFreeList 作为 PageHeap 与 ThreadCache 的中间人，负责将 PageHeap 中的内存切分为小块，在恰当时机分配给 ThreadCache；在释放内存方面，CentralFreeList 获取从 ThreadCache 中回收的内存并在恰当的时机将部分内存归还给 PageHeap。最后一层是

仅供非商业用途或交流学习使用

PageHeap，它是中央堆分配器，被所有线程共享（分配时需要全局锁定），负责与操作系统的直接交互（申请及释放内存），并且大尺寸的内存申请直接通过 PageHeap 进行分配。

第二点，内存分配粒度。如果说线程私有性是从线程角度考虑内存分配的，那么内存分配粒度则是从 TCMalloc 角度考虑了。TCMalloc 有两种粒度（所谓粒度就是管理器可以调度的最小单位）的内存，分别是 object 和 span。span 是连续 page 的内存，而 object 则是由 span 切成的小块。object 的尺寸被预设了一些规格（class），比如 16B、32B 等（截至目前有 88 种不同的规格），同一个 span 切出来的 object 都是相同的规格，如图 15-3 所示。object 不会大于 256KB，超大的内存将直接分配 span 来使用。ThreadCache 和 CentralCache 都是管理 object 的，而 PageHeap 管理的是 span。

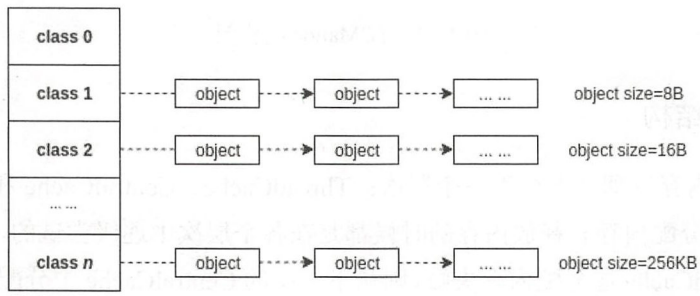


图 15-3　TCMalloc 核心示意图

上面两点组成了 TCMalloc 的核心，这个核心被称为离散式空闲列表算法（Segregated Free List）。在申请小内存（小于 256KB）时，TCMalloc 会根据申请内存的大小，匹配到某个 class 中。举个例子，申请 0～8B 的大小时，会被匹配到 class1 中，分配 8B 大小；申请 9～16B 大小时，会被映射到 class2 中，分配 16 B 大小，以此类推。

从上面的例子可以知道，每个小对象的大小都会被匹配到与之最接近的可分配的 class 中，目前有 88 个可分配的尺寸类别，每个类别的尺寸是不一样的，小一点的 class 尺寸相差 8B，较大的 class 尺寸相差 16B，再大一点的 class 尺寸相差 32B，后面的 class 尺寸也是逐渐增加的。这样一来刚超过上一个 class 尺寸的小对象就会被分配到下一个级别的 class 中，这样可以非常灵活地控制内存，而且极大避免了内存浪费。

15.2.2　小内存分配

继续解释图 15-2，“小内存分配”中对应的 ThreadCache 就是 TCMalloc 中三级分配的第一层，它是一个 TSL（线程本地存储）对象，小于 256KB 的内存申请均由 ThreadCache

仅供非商业用途或交流学习使用

处理（每个线程都独立拥有各自的离散式空闲列表），ThreadCache 分配过程不需要任何锁（没有竞争），极大提高了分配速度。

要知道，TCMalloc 在内存上使用连续虚拟地址，以页（page）为单位、多级缓存进行管理，TCMalloc 把内存块分为两种，其中 span（由地址连续的 page 组成一个大块内存）用于内部管理；而 object（将 span 按照特定大小分割为多个小块，每一个小块可以存储一个对象）用于面向对象分配。

简单来说，分配器会以 page 为单位，向操作系统申请“大块内存”，这些大块内存由 *n* 个地址连续的 page 组成，并用名为 span 的对象进行管理。举个例子，现有一个拥有 128 page 的 span，如果要申请一个 1 page 的 span，就把这个 span 分裂成两个，1 + 127，再把 127 page 的 span 记录下来。

当通过 ThreadCache 分配小内存时，首先会通过 SizeMap 查找要分配的内存所对应的 class 及 object 大小；然后检查空闲列表（free list）是否为空，如果空闲列表（free list）非空，表示线程还有空闲的内存，那么直接从列表中移除第一个 object 并返回，这个过程不需要任何锁，所以速度极快。

如果空闲列表（free list）为空，表示线程内没有空闲的内存了，那么就需要从 CentralFreeList 中获取若干个 object，因为 CentralCache 是被所有线程共享的，至于获取多少个 object 是由慢启动算法决定的，避免浪费内存。取到的 object 会被分配到 ThreadCache 对应的 class 列表中，最终取出其中一个 object 返回。

但若是 CentralFreeList 中的 object 也不够用了，CentralFreeList 就需要向 PageHeap 申请一连串页面（page，由 span 组成，每次会申请 class_to_pages 个页面），申请到的页面（page）会被切割成一系列的 object，再将部分 object 转移给 ThreadCache。

如果 PageHeap 还是不够用，那么就需要向操作系统申请分配更多的内存（以 page 为单位），这就是前面讲到的 sbrk、mmap 等方法，或者通过在/dev/mem 中进行映射。PageHeap 获得更多的 page 之后继续上面的操作，把内存逐级递给线程。

15.2.3　CentralCache

CentralCache 里面有多个 CentralFreeList，即针对每一种 class 的 object。CentralFreeList 和 ThreadCache 不一样，它没有直接维护 object 链表，而是维护 span 的链表，每个 span 下面再挂一个由这个 span 切分出来的 object 的链，如图 15-4 所示。

仅供非商业用途或交流学习使用

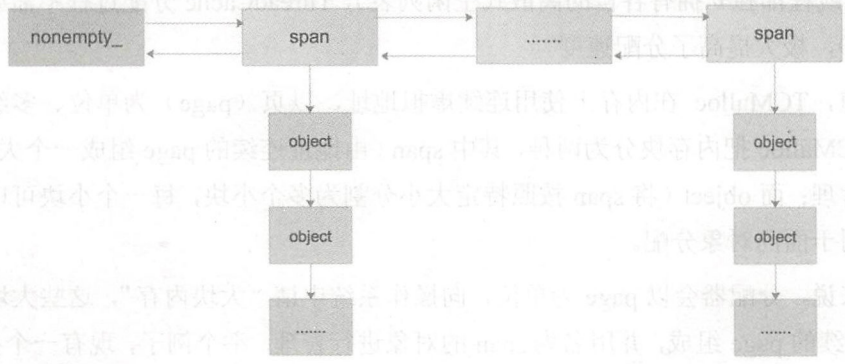


图 15-4　CentralFreeList 结构示意

这样在 span 内的 object 都已经空闲（free）的情况下，将 span 整体回收给 PageHeap（span.refcount_记录了被分配出去的 object 个数）。但是如果每个回收的 object 都需要寻找自己所属的 span，然后才能挂进 freelist，这样就比较耗时了。所以 CentralFreeList 里面还维护了一个缓存（tc_slots_），回收的若干 object 先往缓存里塞，不管 object 大小如何，缓存满了再分类放进相应 span 的 object 链。相反，如果 ThreadCache 申请 object，也是先尝试在缓存里面给，没了再去 span 链那里申请。

那么这个若干具体是多少个 object 呢？其实这是预定义的，称作 batch_size，不同的 class 可能有不同的值。ThreadCache 向 CentralCache 分配或回收 object 时，都尽量以 batch_size 为一个批次。而为了使得缓存简单高效，如果某次分配或者回收的 object 个数小于 batch_size，则会绕过缓存，直接处理。

现在回到图 15-4 上，为了避免在分配 object 时判断 span 是否为空，CentralFreeList 里的 span 链表被分为两个，分别是 nonempty_和 empty_，根据 span 的 objects 链是否有空闲，放入对应链表。当到了需要分配时，只需要在由空变非空、或者由非空变空时移动 span 就可以了。

CentralFreeList 作为整个体系的中间人，它从 PageHeap 中获得 span，并按照预定大小（SizeMap 中的定义）将其分割成大小固定的 object，然后 ThreadCache 可以共享 CentralFreeList 列表。

当 ThreadCache 需要从 CentralFreeList 中获取 object 时，会从 nonempty_链表中获取第一个 span，并从这个 span 的 object 链表中获取可用 object 返回。当该 span 无可用 object

仅供非商业用途或交流学习使用

时，将此 span 从 nonempty_链表移除，并挂到 empty_链表上。

当 ThreadCache 把 object 归还给 CentralFreeList 时，object 会找到它所属的 span，并挂载到 object 链表表头，如果 span 处在 empty_链表，CentralFreeList 会重新将其挂载到 nonempty_链表。

span 里还有一个值用于计算 object 是否已满，每次分配出去一个 object，refcount 值就会加 1，每回收一个 object 就会减 1，如果 refcount 等于 0 就表示此 span 所有 object 都回家了，然后 span 会从 CentralFreeList 的链表中释放，并将其退还给上一层的 PageHeap。

15.2.4　大内存分配

前面我们遇到的内存分配都是一些小于 page 的对象，如果分配的内存大于 page 这个单位，就需要用多个 page 来分配了——把多个 page 组成一个 span 用来分配。

之前说过 PageHeap 是可以向操作系统申请内存的，与大多数现代内存分配器一样，TCMalloc 也是以 page 为单位进行分配的，每次至少向系统申请 1page 的内存空间。TCMalloc 中定义的 page 大小为 8KB。注意一点，目前大部分 Linux 系统都把 1page 的大小定义为 4KB，也就是说 TCMalloc 中的一 page 其实是 Linux 中 2page。

尽管 PageHeap 是按 page 申请内存的，但是它管理内存的基本单位为 span，span 对象其实是地址连续的 page。一个 span 里面至少有 1page 大小（一般都有好多 page）。

下面是 span 的定义，从定义中可以看到 span 的完整结构：

```
struct Span {
  PageID        start;   // span 的内存的起始地址
  Length        length;  // span 里面包含的 page 数量
  Span*         next;    // span 是一种双向链表，因为要被 PageHeap 和 CentralFreeList 使用
  Span*         prev;
  void*         objects; // span 会在 CentralFreeList 中分割为若干 object 组成的 free list
  unsigned int  refcount : 16;  // span 的 object 被引用次数
  unsigned int  sizeclass : 8;  // span 所属的 size class
  unsigned int  location : 2;
  unsigned int  sample : 1;
  // 查询 freelist 中 span 的状态：IN_USE 表示最近是否被使用，状态包括 none、normal 和 returned
  enum { IN_USE, ON_NORMAL_FREELIST, ON_RETURNED_FREELIST };
};
```

那么 PageHeap 是如何组织 span 的呢？

总体来说，PageHeap 维护了一个核心关系：page 与 span 的映射关系。当释放回收一

仅供非商业用途或交流学习使用

个 object 时，把 object 放回原来的位置需要 CentralFreeList 来处理（object 放回原来的 span，然后才还给 PageHeap），但是之所以能够放回对应的 span 里是因为有 page 到 span 的映射关系，地址值经过地址对齐，很容易知道它属于哪一个 page。再通过 page 到 span 的映射关系就能知道 object 应该放到哪里。

span.sizeclass 记录了 span 切分的 object 属于哪个 class，那么属于这个 span 的 object 在释放时就可以放到 ThreadCache 对应 class 的 FreeList 上面，接下来 object 如果要回收还给 CentralFreeList，就可以直接把它挂到对应 span 的 objects 链表上。

page 到 span 的映射关系是基于 radix tree 实现的，你可以把它理解为一种很大的数组，用 page 值作为偏移可以访问到 page 所对应的 span（也有多个 page 指向同一个 span 的情况，因为 span 有时可不止一个 page）。查询 radix tree 需要消耗一定时间，所以为了避免这些开销，PageHeap 和 CentralFreeList 类似，维护了一个最近活跃的 page 到 class 对应关系的缓存。为了保持缓存的效率，缓存只有 64KB，旧的对应关系会被新来的对应关系替换掉。

当需要某个尺寸的 span 没有空闲时，可以把更大尺寸的 span 拆分，如果大的 span 也没有了，就是向操作系统要的时候了；回收时，也需要判断相邻的 span 是否空闲，以便将它们组合。判断相邻 span 还是使用 radix tree 查询，这种数据结构就像一个大数组，可以获取当前 span 前后相邻的 span 地址。span 的尺寸有从 1 page 到 255 page 的所有规格，所以 span 可以以 page 为单位，用任意尺寸进行拆分和组合。

上面是 PageHeap 组织 span 的大体情况，实际上从 PageHeap 的结构上更容易看出它管理 span 的方式：

```
PageMap pagemap_;
struct SpanList {
  Span       normal;
  Span       returned;
};
SpanList large_;
SpanList free_[kMaxPages];   // kMaxPages = 128
```

以上这段代码中有两个 SpanList，PageHeap 就是通过这两个数据结构管理所有的 span（free_数组和 large_列表），free_数组存放 size 小于 kMaxPages 的 span，按 page 数目区分；而 large_列表存放大于等于 kMaxPages 的 span，不再按照 page 数目区分。

这两个数据结构内部的 span 又分为 normal 和 returned 两个链表以区分活跃和不活跃

仅供非商业用途或交流学习使用

的内存，其中 normal 包含的 span，是页面明确映射到进程地址空间的 span；returned 包含的 span，是 TCMalloc 已经通过 madvise 归还给操作系统的空间，调用 madvise 相当于取消了虚拟内存与物理内存的映射。

PageHeap 本身并不会将内存释放给操作系统，因为它们之间的交互都是针对一批连续 page 的，要想回收到整批的 page，基本不可能。在 PageHeap 里面，不活跃的内存会放到 returned 中，与 normal 隔离。活跃的 normal 上的内存总是被优先使用，所以操作系统不会搭理它们。至于 returned 里面的内存，意味着是不活跃的，操作系统在内存不够的时候会优先将它们回收。

其实没有 returned 链表也可以做到活跃与不活跃内存的区分，反正 normal 也是一个链表，span 按活跃度排序就好了，链表尾部的 span 意味着长期不被使用，是可以回收的，这样看起来不是更省事了吗？然而事实上，span 进入 returned 时，TCMalloc 还做了一件事，那就是调用 madvise(MADV_DONTNEED)告诉操作系统这些内存已经暂时不需要了，操作系统可以随便处理。

最终结果是虽然通过 madvise“表面归还”给了操作系统，但是操作系统有可能还没有来得及收回这部分内存，所以 TCMalloc 留着 returned 列表，以备后续可以再次直接利用而不必向操作系统申请。如果操作系统回收之前，TCMalloc 再次使用了这些 page，那操作系统就暂时不会再回收这些内存空间（如果“不幸”被操作系统回收了，TCMalloc 还想用的时候，Linux 内核会引发 page fault，并将其映射到一块全零的内存空间，虽然不影响使用，但是会影响性能）。

所以在分配内存时，PageHeap 首先判断请求的 page 数量大小，如果小于 kMaxPages，那么先从 free_数组中找，从要求大小的位置开始往后找，先找 normal 链表再找 returned 链表。如果在 normal 链表中找到且找到的 span 状态为 Span::ON_NORMAL_FREELIST，那么直接从里面切割需要的 span 返回给 CentralCache。如果在 returned 链表中找到且找到的 span 状态为 Span::ON_RETURNED_FREELIST，那么直接从里面切割需要的 span 返回给 CentralCache。

但是若需要的 page 数量大于 kMaxPages，那么将从 large_列表中找。同样先从 normal 链表入手，然后再找 returned 链表，找到最符合大小且状态为空闲的最小的 span，然后切割出来返回给 CentralCache。

仅供非商业用途或交流学习使用



如果在 large_列表中都没有找到合适的 span，就需要通过 GrowHeap 增加 Heap 的大小了，通过向系统申请内存并包装成 span，然后存入 heap 中，才能进行分配。

15.3 Mspan（内存管理器）

前面提到过，Go 语言内存管理器是基于 TCMalloc 实现的，底层直接调用 mmap 函数，并使用 bestfit 进行动态分配，但 Go 语言的内存管理器也做了一些优化，所以和 TCMalloc 并不完全一致。在 Go 语言中，内存分为两部分，一部分作为堆，供内存分配；另一部分作为 bitmap，用来管理堆。这两部分的内存都是从同一个地址开始申请的，向高地址的方向增长的就是内存池，向低地址方向增长的就是 bitmap。

下面主要介绍内存管理缓存结构。

Go 语言为每个系统线程分配了一个本地 MCache（类似 TCMalloc 中的 ThreadCache，不过 Go 语言改了名称），少量的地址分配就是从 MCache 分配的，并且定期进行垃圾回收，所以 Go 语言的分配器包含了显式与隐式的调用。Go 语言定义的小块内存与 TCMalloc 基本一致，Go 语言底层会把这些小块内存按照指定规格（和 TCMalloc 的 class 类似）进行切割。整个过程结构都与 TCMalloc 相似，不再赘述。

下面是 Go 语言内存分配的主要组件（如图 15-5 所示）。

- MCache：每个尺寸的 class 都有一个空闲链表。每个 goroutine（线程）都有自己的局部 MCache（小对象从它取，无须加锁，没有竞争，因此十分高效）。
- MCentral：与 TCMalloc 的 CentralCache 类似，MCache 可以从这里获取更多内存，当自身无空闲内存时，可以向 MHeap 申请一个 span（只能一个），申请的 span 包含多少个 page 由 central 的 sizeclass 确定。
- MHeap：负责将 MSpan 组织和管理起来。分配过程和 TCMalloc 类似，从 free 数组中分配，如果发生切割则将剩余的部分放回 free 数组中。回收过程也类似，回收一个 Mspan 时，先查找它的相邻地址，再通过 map 映射得到对应的 Mspan，如果 Mspan 的状态是未使用，则可以将两者合并。最后将这个 page 或者合并后的 page 归还到 free 数组分配池或者 large 中。

仅供非商业用途或交流学习使用

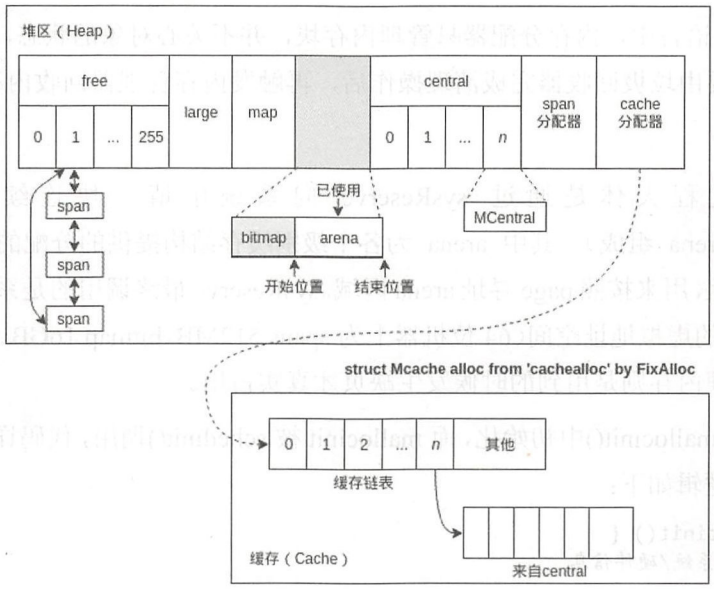


图 15-5　Go 语言 alloc 示意图

可以看到 Go 语言的内存分配结构和 TCMalloc 是基本一致的，需要注意的是图 15-5 中的一句话：struct Mcache alloc from 'cachealloc' by FixAlloc，意思是说用户进程 newobject 是从图 15-5 所示的 arena 区域分配的，而 runtime 层自身管理的结构（比如 mcache 等）是专门设计了 fixAlloc 来分配的，这和 TCMalloc 非常不一样。

在 Go 语言里面，内存分配有三个基本的概念要清楚，即 G-M-P：G 代表 Goroutine 执行的上下文环境；M 代表操作系统线程；P 代表 Processer（调度器），这是进程调度的关键，调度器可以认为“约等于”CPU。一个 Goroutine 的运行需要 G-P-M 三部分结合起来。

Go 语言的内存分配器与 TCMalloc 库基本一样，是一个带内存池的分配器，底层直接调用操作系统的 mmap 等函数。内存池是一种内存分配方式。在通常情况下，我们所理解的分配内存都是通过直接使用 new、malloc 等 API 申请的，由于所申请内存块的大小不定，当频繁使用时会造成大量的内存碎片并进而降低性能。

于是就有了内存池的方法，简单来说，内存池是在真正使用内存之前，先申请一大块内存（例如 1MB），并把分配到的内存划分为一定数量的、大小相等（一般而言）的小内存块构成链表，留作备用。当有新的内存需求时，就从内存池中分出一小部分内存块即可。若内存块不够再继续申请新的内存，如果内存闲置过多则将部分内存归还给操作系统。这样做的一个显著优点是，使得内存分配效率得到提升。

仅供非商业用途或交流学习使用



不过在 Go 语言中，内存分配器只管理内存块，并不关心对象的状态，而且不会主动回收内存，需要由垃圾回收器完成清理操作后，再触发内存管理器回收内存。

1. 初始化

初始化过程大体是通过 sysReserve 向系统申请一块连续的内存（由 spans+bitmap+arena 组成）。其中 arena 为各个级别缓存结构提供的分配的内存块，spans 是一个指针数组，用来按照 page 寻址 arena 区域。sysReserve 最终调用的是系统函数 mmap，会申请 512GB 的虚拟地址空间（64 位机器上为 spans 512MB、bitmap 16GB、arena 512GB），当然真正的物理内存则是用到的时候发生缺页才真实占用。

MHeap 在 mallocinit()中初始化，而 mallocinit 被 schedinit()调用，代码详见/src/runtime/proc.go，大体逻辑如下：

```
func mallocinit() {
    // 检查系统/硬件信息
    ... ...
    // 初始化规格 class 和 size 的对照方法
    initSizes()
    // 计算预留空间大小
      // 系统指针大小 PtrSize = 8，表示这是一个 64 位系统
      /* spans 记录的是 arena 区的块页号和对应的 mspan 指针的对应关系。比如 arena 区内存地址 x，对应的页号就是 page_num = (x - arena_start) / page_size，那么 spans 就会记录 spans[page_num] = x。如果 arena 为 512 GB，spans 区的大小为 512 GB / 8KB * 8 = 512 MB。这里值得注意的是 Go 语言的内存管理虚拟地址页大小为 8KB */
    if ptrSize == 8 && (limit == 0 || limit > 1<<30) {
      arenaSize := round(_MaxMem, _PageSize)
      bitmapSize = arenaSize / (ptrSize * 8 / 4)
      spansSize = arenaSize / _PageSize * ptrSize
      pSize = bitmapSize + spansSize + arenaSize + _PageSize
      p1 = uintptr(sysReserve(unsafe.Pointer(p), pSize, &reserved))
    }
    // 尝试预留地址（区分不同平台）
    for i := 0; i <= 0x7f; i++ {
      ...
      pSize = bitmapSize + spansSize + arenaSize + _PageSize
      // 向 OS 申请大小为 pSize 的连续的虚拟地址空间
      p = uintptr(sysReserve(unsafe.Pointer(p), pSize, &reserved))
    }

    // 初始化部分 mheap 中变量
    p1 := round(p, _PageSize)
    spansStart := p1
    mheap_.bitmap = p1 + spansSize + bitmapSize
    mheap_.arena_start = p1 + (spansSize + bitmapSize)
```

仅供非商业用途或交流学习使用




```
    mheap_.arena_end = p + pSize
    mheap_.arena_used = p1 + (spansSize + bitmapSize)
    mheap_.arena_reserved = reserved

    // 其他部分初始化，67 个 mcentral 在这里初始化
    mheap_.init(spansStart, spansSize)
    //获取当前 G
    _g_ := getg()
    // 获取 G 上绑定的 M 的 mcache
    _g_.m.mcache = allocmcache()
}
```

MHeap 初始化相关指针，使之可以在 arena 这块内存寻址，同时初始化 cachealloc 这个固定分配器，最后执行的 m.mcache = allocmcache()是每个 goroutine 创建时都要初始化的。此时才真正创建了 mcache，并且初始化 mcache 里整个数组对应的 mspan 为 emptyspan。

在上面的代码中，p 是从连续虚拟地址的起始地址先进行对齐，然后初始化 arena、bitmap、spans 地址。mheap_.init()会初始化 fixalloc 等相关的成员，还有 MCentral 的初始化。

MCentral 的初始化比较简单，设置自身级别并将两个 mspanList 初始化。而 MCache 在 procresize(nprocs int32) *p 中初始化（代码如下），procresize 也在 schedinit()中调用，顺序在 mallocinit()之后，也就是说发生在 MHeap 与 MCentral 的初始化后面。

```
func procresize(nprocs int32) *p {
    ...
    // 初始化新的 P（调度器）
    for i := int32(0); i < nprocs; i++ {
        pp := allp[i]
        if pp == nil {
            pp = new(p)
            pp.id = i
            pp.status = _Pgcstop
            pp.sudogcache = pp.sudogbuf[:0]
            for i := range pp.deferpool {
                pp.deferpool[i] = pp.deferpoolbuf[i][:0]
            }
            atomicstorep(unsafe.Pointer(&allp[i]), unsafe.Pointer(pp))
        }
        // P mcache 初始化
        if pp.mcache == nil {
            if old == 0 && i == 0 {
                if getg().m.mcache == nil {
                    throw("missing mcache?")
                }
```

仅供非商业用途或交流学习使用




```
                // P[0] 分配给主 goroutine
                pp.mcache = getg().m.mcache // bootstrap
            } else {
                // P[0] 之外的 P 申请 mcache
                pp.mcache = allocmcache()
            }
        }
        ...
    }
    ...
}
```

所有的 P 都存放在一个全局数组 allp 中，procresize()的目的就是将 allp 用到的 P 进行初始化，同时对多余的 P 的资源隔离。至此，管理结构 MHeap、MCentral 及每个 P 的 MCache 都初始化完毕，接下来进入分配阶段。

2. 分配

分配的整个流程是：将小对象所需内存大小向上取整到最近的尺寸类别或者称为规格（class），查找相应的 MCache 的空闲链表，如果链表不空，直接从上面分配一个对象，这个过程不加锁；如果 MCache 空闲链表是空的，通过 MCentral 的空闲链表取一些对象进行补充；如果 MCentral 的空闲链表也是空的，则在 MHeap 中取用一些 page 对 MCentral 进行补充，然后将这些内存分割成特定规格；如果 MHeap 没有足够大的 page 时，从操作系统分配一组新的 page。

为了避免逃逸的情况，假设关闭了内联优化，现在来看源码，当 new 一个对象时，调用的是 newobject()，但实际上调用的是 mallocgc()：

```
func mallocgc(size uintptr, typ *_type, needzero bool) unsafe.Pointer {
    c := gomcache()
    if size <= maxSmallSize {
        // size 小于 16B 的不用扫描的对象，直接从 mcache 的 tiny（极小）上分
        if noscan && size < maxTinySize {
            off := c.tinyoffset
            if off+size <= maxTinySize && c.tiny != 0 {
                x = unsafe.Pointer(c.tiny + off)
                return x
            }
            // 若没有 tiny 了，则从 mcache 中相应规格（class size）的 mspan 中查找
            span := c.alloc[tinySizeClass]
            v, _, shouldhelpgc = c.nextFree(tinySizeClass)
            x = unsafe.Pointer(v)
        } else {
            // 普通小于 4KB 小对象先计算规格（class size）
```

仅供非商业用途或交流学习使用




```
            span := c.alloc[sizeclass]
            v, span, shouldhelpgc = c.nextFree(sizeclass)
        }
    } else {
        // 大对象直接从 heap 分配 span
        systemstack(func() {
            s = largeAlloc(size, needzero)
        })
        x = unsafe.Pointer(s.base())
    }
    return x
}
func (c *mcache) nextFree(sizeclass uint8) (v gclinkptr,
s *mspan, shouldhelpgc bool) {
    s = c.alloc[sizeclass]
    freeIndex := s.nextFreeIndex()
    if freeIndex == s.nelems {
        systemstack(func() {
            c.refill(int32(sizeclass))
        })
        s = c.alloc[sizeclass]
        freeIndex = s.nextFreeIndex()
    }
    v = gclinkptr(freeIndex*s.elemsize + s.base())
    return
}
```

上面代码省略了很多信息，为了更清晰地了解这个过程，根据对象的大小分为 3 个级别：tiny（极小）、小、大，首先是 tiny（小于 16B）级别的：

```
off := c.tinyoffset
// 地址对齐
if size&7 == 0 {
    off = round(off, 8)
} else if size&3 == 0 {
    off = round(off, 4)
} else if size&1 == 0 {
    off = round(off, 2)
}

// 若之前 tiny 剩余空间够用，则将极小对象拼在一起
if off+size <= maxTinySize && c.tiny != 0 {
    x = unsafe.Pointer(c.tiny + off)
    c.tinyoffset = off + size
    c.local_tinyallocs++
    mp.mallocing = 0
    releasem(mp)
    return x
```

```
}

// tiny 不够了，为其重新分配一个 16B 的内存块
// Allocate a new maxTinySize block.
span := c.alloc[tinySizeClass]
v := nextFreeFast(span)
if v == 0 {
    v, _, shouldhelpgc = c.nextFree(tinySizeClass)
}
x = unsafe.Pointer(v)
// 将申请的内存块全置为 0
(*[2]uint64)(x)[0] = 0
(*[2]uint64)(x)[1] = 0

// 新申请的剩余空间大于之前的剩余空间
// 如果申请的内存块用不完，则将剩下的给 tiny，用 tinyoffset 记录分配了多少
if size < c.tinyoffset || c.tiny == 0 {
    c.tiny = uintptr(x)
    c.tinyoffset = size
}
size = maxTinySize
```

对于小于 16B 的内存块，Mcache 有一个专门的内存区域“tiny”用来分配，tiny 指针指向 tiny 内存块的起始地址。如上所示，tinyoffset 表示 tiny 当前分配的地址位置，之后的分配根据 tinyoffset 寻址。先根据要分配的对象大小进行地址对齐，比如 size 是 8 的倍数，tinyoffset 就和 8 对齐，然后进行分配。如果 tiny 剩余的空间不够用，则重新申请一个 16B 的内存块，并分配给 object。如果有余，则记录在 tiny 上。

接着是大对象（大于 32KB），小对象有些复杂一会再说，对于大于 32KB 的内存分配，直接跳过 mcache 和 mcentral，通过 mheap 分配。大于 32KB 的内存分配都是分配整数页，先右移然后低位与计算需要的页数。

```
func mallocgc(...) {
    } else {
        var s *mspan
        shouldhelpgc = true
        systemstack(func() {
            // largeAlloc 会调用 mheap_.alloc，这个方法在小对象申请时已经追踪过
            s = largeAlloc(size, needzero)
        })
        s.freeindex = 1
        s.allocCount = 1
        x = unsafe.Pointer(s.base())
        size = s.elemsize
    }
```

```
    ...
}

func largeAlloc(size uintptr, needzero bool) *mspan {
    ...
    npages := size >> _PageShift
    if size&_PageMask != 0 {
        npages++
    }
    ...
    s := mheap_.alloc(npages, 0, true, needzero)
    if s == nil {
        throw("out of memory")
    }
    s.limit = s.base() + size
    heapBitsForSpan(s.base()).initSpan(s)
    return s
}
```

最后是对于大小介于 16KB～32KB 的小对象内存分配，首先计算应该分配的 sizeclass，然后去 mcache 里面申请，如果不够，就让 mcache 向 mcentral 申请再分配。Mcentral 为 mcache 分配完之后会判断自己需不需要扩充，如果需要就向 mheap 申请。

这个过程已经说了很多次了，下面来看具体源码，计算应该分配的 sizeclass 代码如下：

```
var sizeclass uint8
if size <= smallSizeMax-8 {
    sizeclass = size_to_class8[(size+smallSizeDiv-1)/smallSizeDiv]
} else {
    sizeclass = size_to_class128[(size-smallSizeMax+largeSizeDiv-1)/largeSizeDiv]
}
/*
数组 size_to_class8 的第 i 个值表示地址区间[(i-1)*8,i*8]对应的 sizeclass，这里 smallSizeDiv
等于 8。size_to_class128 类似。
小于 1024-8=1016（smallSizeMax=1024），则使用数组 size_to_class8，否则使用数组 size_to_
class128。

举个例子，比如要分配 17B 的内存 （16B 以下的使用 mcache.tiny 分配）：
sizeclass = size_to_calss8[(17+7)/8] = size_to_class8[3] = 3
*/
size = uintptr(class_to_size[sizeclass])
span := c.alloc[sizeclass]
// 从对应的 span 里面分配一个 object
v := nextFreeFast(span)
if v == 0 {
    v, span, shouldhelpgc = c.nextFree(sizeclass)
}
x = unsafe.Pointer(v)
```

```
if needzero && span.needzero != 0 {
    memclrNoHeapPointers(unsafe.Pointer(v), size)
}
```

计算出 sizeclass，那么就可以去 mcache.alloc[sizeclass]分配了，注意这是一个 mspan 指针，这里有两个重要的方法 nextFreeFast 和 nextFree，它们才是真正执行分配的工作。

```
// 如果下一个对象快速可用，则 nextFreeFast 返回下一个空闲对象，否则返回 0
func nextFreeFast(s *mspan) gclinkptr {
    // 计算 s.allocCache 从低位起有多少个 0
    theBit := sys.Ctz64(s.allocCache)
    if theBit < 64 {
        result := s.freeindex + uintptr(theBit)
        if result < s.nelems {
            freeidx := result + 1
            if freeidx%64 == 0 && freeidx != s.nelems {
                return 0
            }
            // 更新位图、可用游标
            s.allocCache >>= (theBit + 1)
            s.freeindex = freeidx
            // 根据 result 和 s.elemsize 起始地址计算 v
            v := gclinkptr(result*s.elemsize + s.base())
            s.allocCount++
            return v
        }
    }
    return 0
}
```

在以上代码中，allocCache 用位图表示内存是否可用，1 表示可用，然后通过 span 里面的 freeindex 和 elemsize 来计算地址。现在的重点是如果 mcache.alloc[sizeclass]不够用了，则通过 nextFree 向 mcentral（甚至 mheap）申请内存，看下面代码：

```
func (c *mcache) nextFree(sizeclass uint8) (v gclinkptr, s *mspan, shouldhelpgc boo
l) {
    s = c.alloc[sizeclass]
    shouldhelpgc = false
    freeIndex := s.nextFreeIndex()
    if freeIndex == s.nelems {
        // span 为空

if uintptr(s.allocCount) != s.nelems {
            println("runtime: s.allocCount=", s.allocCount, "s.nelems=", s.nelems)
            throw("s.allocCount != s.nelems && freeIndex == s.nelems")
        }
      // 重新填充当前的 mcache
        systemstack(func() {
```

```
            // 此处 mcache 向 mcentral 申请
            c.refill(int32(sizeclass))
        })
        shouldhelpgc = true
        s = c.alloc[sizeclass]
        // mcache 向 mcentral 申请完之后，再次从 mcache 申请
        freeIndex = s.nextFreeIndex()
    }
    ...
}
```

从上面的代码中可以看到，如果 nextfree 在 mcache 相应规格的 mspan 里拿不到 object，那么就需要从 mcentral 中“再次装填”（refill）内存块。

mcache 向 mcentral 申请内存实际是通过 refill 来实现的。你可能还记得一个细节：没有可用 object 的 MSpan 是要还给 MCentral 的，并挂到 MCentral 的 empty 链表中，在这里相应体现为把没有 object 的 MSpan 的 incache 值设置为 false，等待 sweep 回收。

sweepgen 是一个回收标记，其中 sg-2 表示等待回收，sg-1 表示正在回收，sweepgen 表示已经回收。从 MCentral 中获取 MSpan 时有可能遇到当前的 SPan 正等待或正被回收，所以可以先把等待回收的 MSpan 返回，用来再次填装（refill）MCache，因此将它插入 empty 链表中：

```
func (c *mcache) refill(sizeclass int32) *mspan {
    _g_ := getg()
    _g_.m.locks++
    // 向 mcentral 归还当前的 mspan
    s := c.alloc[sizeclass]
    if uintptr(s.allocCount) != s.nelems {
        throw("refill of span with free space remaining")
    }
    if s != &emptymspan {
      // 空 Span 回收标志
        s.incache = false
    }
    // 向 mcentral 申请，注意 mcentral.cacheSpan()
    s = mheap_.central[sizeclass].mcentral.cacheSpan()
    ...
    c.alloc[sizeclass] = s
    _g_.m.locks--
    return s
}

// 分配一个 span 以在 MCache 中使用
func (c *mcentral) cacheSpan() *mspan {
```

```
    sg := mheap_.sweepgen
retry:
    var s *mspan
      // 先从非空列表中找
    for s = c.nonempty.first; s != nil; s = s.next {
        if s.sweepgen == sg-2 && atomic.Cas(&s.sweepgen, sg-2, sg-1) {
            // 等待回收，可以返回使用
            c.nonempty.remove(s)
            c.empty.insertBack(s)
            s.sweep(true)
            goto havespan
        }
        if s.sweepgen == sg-1 {
            // 正在回收，忽略
            continue
        }
        c.nonempty.remove(s)
        c.empty.insertBack(s)
        goto havespan
    }
      // 没有则从空列表中找
    for s = c.empty.first; s != nil; s = s.next {...}
    // 实在没有再申请
      s = c.grow()
    c.empty.insertBack(s)
havespan:
    ...
    return s
}
/*
到这里，如果内存还不够就需要让 MCentral 向 MHeap 申请内存了
*/
// 此处由 MCentral 申请
func (c *mcentral) grow() *mspan {
    npages := uintptr(class_to_allocnpages[c.sizeclass])
    size := uintptr(class_to_size[c.sizeclass])
    n := (npages << _PageShift) / size
    // 这里向 MHeap 申请
    s := mheap_.alloc(npages, c.sizeclass, false, true)
    ...
    return s
}
```

至此，上面几个函数的调用链已经到了 MCentral 都没有内存的地步，需要向 MHeap 申请了，grow 会计算出当前所需规格对应的 page 数目，然后从 MHeap 中的 nPage 取一个 mspan。前面提到的 free 数组是一个指针数组，每个指针对应一个 MSpan 的链表，数组按照 nPage 寻址。

若所需的 nPage 链表中都没有空闲 MSpan，则 MHeap 也需要补充内存了。

```
func (h *mheap) alloc(npage uintptr, sizeclass int32, large bool, needzero bool)
*mspan {
    ...
    systemstack(func() {
        s = h.alloc_m(npage, sizeclass, large)
    })
    ...
    return s
}

func (h *mheap) alloc_m(npage uintptr, sizeclass int32, large bool) *mspan {
    ...
    s := h.allocSpanLocked(npage)
    ...
    return s
}

// Best-fit 算法
func (h *mheap) allocSpanLocked(npage uintptr) *mspan {
    // 先从 128 页（1MB）以内的空闲列表中寻找
    for i := int(npage); i < len(h.free); i++ {
        list = &h.free[i]
        ...
    }

    // Best-fit 对于大对象申请也会用到这个方法
    // 基本思路是找到最小可以满足需求的 MSpan，如果有多个，选择地址最小的
    list = &h.freelarge
    s = h.allocLarge(npage)
    if s == nil {
        // 如果 MHeap 也没有空间就向系统申请
        if !h.grow(npage) {
            return nil
        }
        s = h.allocLarge(npage)
        if s == nil {
            return nil
        }
    }

HaveSpan:
    // 转移 s
    list.remove(s)
    if s.inList() {
        throw("still in list")
    }
```

```
    // 对于申请到的内存大于想要的，将其拆分，避免浪费
    if s.npages > npage {
        ...
        h.freeSpanLocked(t, false, false, s.unusedsince)
        s.state = _MSpanFree
    }

    return s
}

// 向系统申请空间
/* MHeap 增长会通过 h.sysAlloc（代码在下方）直接向 arena 区域申请 nB 的内存，n 具体值根据
nPage 大小计算。此时，arena 区域的一些指针标记开始移动，最终将 MSpan 加入链表，等待分配 */
func (h *mheap) grow(npage uintptr) bool {
    // 计算页数
    npage = round(npage, (64<<10)/_PageSize)
    ask := npage << _PageShift
    if ask < _HeapAllocChunk {
        ask = _HeapAllocChunk
    }

    v := h.sysAlloc(ask)
    s := (*mspan)(h.spanalloc.alloc())
    s.init(uintptr(v), ask>>_PageShift)
    p := (s.base() - h.arena_start) >> _PageShift
    for i := p; i < p+s.npages; i++ {
        h.spans[i] = s
    }
    atomic.Store(&s.sweepgen, h.sweepgen)
    s.state = _MSpanInUse
    h.pagesInUse += uint64(s.npages)
    // 加入链表
    h.freeSpanLocked(s, false, true, 0)
    return true
}

func (h *mheap) sysAlloc(n uintptr) unsafe.Pointer {
  ...
  // sysReserve 调用 mmap 预留空间，至此调用链结束
  p := uintptr(sysReserve(unsafe.Pointer(h.arena_end), p_size, &reserved))
  ...
}
```

整个函数调用链如上所示，最后 sysAlloc 会调用系统函数（mmap 之类的）向操作系统申请内存，至此 Go 语言的内存分配流程全部结束，再往后就是操作系统内核的世界了，不再归 Go 语言内存分配器管理。

3. 回收释放

这里的回收并非是垃圾回收，而是更简单的内存回收。还记得上面说到的 sweepgen 标记吧！简单来说，MSpan 里有 sweepgen 回收标记，回收的内存会先全部回到 MCentral，如果已经回收所有的 MSpan，就还给 MHeap 的空闲列表。回收内存的一个很重要的原因是为了复用，所以很多时候并不会直接释放内存。

先说 MCache 吧！使用内存时有两种情况：第一种是用完了闲置，第二种是用了但没用完，前者直接标记等待回收就可以，至于多出来没用到的部分就需要另外想办法还给 MCentral 了：

```
// 检查空闲的 MCache
func freemcache(c *mcache) {
    systemstack(func() {
        c.releaseAll()
        ...

        lock(&mheap_.lock)
        purgecachedstats(c)
        mheap_.cachealloc.free(unsafe.Pointer(c))
        unlock(&mheap_.lock)
    })
}
/*
releaseAll() 负责将 mcache.alloc 中各个 sizeclass 中的 MSpan 归还给 MCentral。
此处的归还操作需要加锁，因为 MCentral 是全局的
*/
func (c *mcache) releaseAll() {
    for i := 0; i < _NumSizeClasses; i++ {
        s := c.alloc[i]
        if s != &emptymspan {
            mheap_.central[i].mcentral.uncacheSpan(s)
            c.alloc[i] = &emptymspan
        }
    }
    // 清除 tinyalloc 池
    c.tiny = 0
    c.tinyoffset = 0
}
```

用完那部分，通过 sweep 标记等待回收：

```
func (s *mspan) sweep(preserve bool) bool {
    res = mheap_.central[cl].mcentral.freeSpan(s, preserve, wasempty)
}
func (c *mcentral) freeSpan(s *mspan, preserve bool,
```

```
wasempty bool) bool {
    if wasempty {
        c.empty.remove(s)
        c.nonempty.insert(s)
    }
  ...
    c.nonempty.remove(s)
    mheap_.freeSpan(s, 0)
    return true
}
```

sysmon 是监控线程，它会遍历 MHeap 中 large 列表里的所有空闲的 MSpan，发现空闲时间超过阈值就调用 madvise，让系统内核释放这个线程相关的物理内存。

经过上面的步骤，MCache 的空闲 MSpan 已经还给 MCentral 了，接下来就是 MCentral 还给 MHeap 了，这个过程简单来说是当 MSpan 的 object 全部收回时，将 MSpan 归还给 Mheap：

```
// 检查 MSpan 是否空闲
func (c *mcentral) freeSpan(s *mspan, preserve bool, wasempty bool) bool {
    ...
    lock(&c.lock)
    ...
    if s.allocCount != 0 {
        unlock(&c.lock)
        return false
    }

    c.nonempty.remove(s)
    unlock(&c.lock)
    mheap_.freeSpan(s, 0)
    return true
}
```

到了最后，MHeap 那里并不会定时向操作系统归还内存，而是先把相邻的 span 合并，使之成为一块更大的内存，以 page 为单位调度回收。

15.4 垃圾回收

与 C++通过指针引用计数来回收对象不一样，Go 语言的垃圾回收是基于标记清理算法的。由于引用计数不能处理循环引用，为了避免引用计数的缺陷，后来出现了各种垃圾回收机制，包括标记清理、节点复制、分代收集等垃圾回收算法。

Go 语言官方形容 Go 语言的垃圾回收为：

- 并发标记和清理（concurrent mark and sweep）。
- 写屏障（write barrier）。
- 非分代（non-generational）。
- 非紧缩（non-compacting）。

顾名思义，标记清理算法就是将需要的内存块进行标记（mark），没有标记的内存块将被清理（sweep）。

15.4.1　标记清理算法

标记清理算法是一个很古老（20 世纪 70 年代提出）且很基础的垃圾回收算法，该算法中有一个标记初始的 root 区域，以及一个受控堆区。root 区域主要是程序运行到当前时刻的栈和全局数据区域。在受控堆区中，很多数据是程序以后不需要用到的，这类数据就可以被当作垃圾回收了。

判断一个对象是否是垃圾，需要看这个对象是否被当前栈或全局数据区域（就是 root 区域）的对象直接或间接地引用。如果没有任何对象引用到它，则说明它没有被使用，因此可以安全地当作垃圾回收。

标记清理算法分为两阶段：标记阶段和清扫阶段。标记阶段，从 root 区域出发，扫描所有 root 区域的对象直接或间接引用到的对象，将这些对象全部加上标记。在回收阶段，扫描整个堆区，对所有无标记的对象进行回收。

这里会引出几个问题：怎样实现并发标记？标记记录在哪里？怎样知道对象是否被引用？什么时候触发清理动作？回收时进程怎么办？

1．如何标记

一直以来 Go 语言的垃圾回收都是争议比较大的地方，不止 Go 语言，其他语言也是，垃圾回收永远是一门大学问。在 Go 1.5 之前，垃圾回收的标记和清理都是 STW（Stop The World），就是在标记和清理时都要停止所有的 goroutine，以此来保证已经被标记的区域不会被再次修改引用关系，造成清理错误。

这种方式在 1.5 版本之后的 Go 语言里得到了改进，毕竟每次垃圾回收都要 Stop The World 显然太低效率了。从 Go 1.5 开始采用三色标记法实现标记阶段的并发（清理阶段目

仅供非商业用途或交流学习使用

前到 Go 1.8 为止都还是需要 STW）。

三色标记的步骤是：最开始所有对象都是白色；扫描所有可达对象，标记为灰色，放入待处理队列；从队列里提取灰色对象，将其引用对象标记为灰色放入队列，自身标记为黑色；写屏障，监控对象内存修改，重新标色或放入队列；完成标记后对象不是白色就是黑色，清理操作只需把白色对象回收即可。

所谓并发标记，一是指通过 write-barrier（写屏障）能够与用户代码并发进行，二是指通过 gc-work 队列实现非递归地标记可达对象，换而言之标记工作不是递归进行的，而是多个 goroutine 并发进行的。

为什么需要 write-barrier？很简单，对于和用户程序并发运行的垃圾回收算法，用户程序会一直修改内存，所以需要记录下来。写屏障的过程是当发现对象已经标记为黑色了，但该对象引用的对象却变了，那么把后来引用的对象变灰入队，原来的被引用对象保持灰色不变。这个 write_barrier 是编译器在每一处内存写操作前生成一小段代码来做的。

接下来是并发标记，要实现非递归地遍历标记可达节点，就需要一个队列。这个队列还可以有助于区分黑色对象和灰色对象，因为标记位只有一个。在队列中的标记是灰色对象，标记了但是不在队列中的是黑色对象，未标记的是白色对象。

下面是实现源码：

```
// 每个 markWorker 都执行 gcDrain 这个标记过程
func gcDrain(gcw *gcWork, flags gcDrainFlags) {
    // 如果还没有 root 区域入队则 markroot
    markroot(gcw, job)
    if idle && pollWork() {
        goto done
    }
    // 节点出队
    b = gcw.get()
    scanobject(b, gcw)
done:
}
func scanobject(b uintptr, gcw *gcWork) {
    hbits := heapBitsForAddr(b)
    s := spanOfUnchecked(b)
    n := s.elemsize
    for i = 0; i < n; i += sys.PtrSize {
        // 找到这个字（指的是最小内存管理单位）对应的位地址（bits）并查询其类型值是否为指针
        if bits&bitPointer == 0 {
            continue // 如果不是指针则跳出循环
        }
```

仅供非商业用途或交流学习使用




```
....
        // 标记这个对象
        if obj, hbits, span, objIndex := heapBitsForObject(obj, b, i); obj != 0 {
            greyobject(obj, b, i, hbits, span, gcw, objIndex)
        }
    }
    gcw.bytesMarked += uint64(n)
    gcw.scanWork += int64(i)
}
func greyobject(obj, base, off uintptr, hbits heapBits,
     span *mspan, gcw *gcWork, objIndex uintptr) {
    mbits := span.markBitsForIndex(objIndex)
    // 如果已经标记，则什么都不做
    if mbits.isMarked() {
        return
    }
    if !hbits.hasPointers(span.elemsize) {
        return
    }
    gcw.put(obj)
}
```

在以上代码中，gcStart 是 GC 的入口函数，根据 gcMode 做处理，它准备了 N 个 goMarkWorkers，每个 worker 都处理以下相同流程：

- 如果是第一次标记，则首先调用 markroot 将所有 root 区域的指针入队。
- 从 gcw（gc-work 队列）中取节点，开始扫描处理（scanobject），节点出队列就是黑色了。
- 扫描时获取该节点所有子节点的类型信息，判断是不是指针，如果是指针且没有被标记则标记灰色（greyobject）入队。
- 每个 worker 都去 gcw 中拿任务直到队列为空。

2. 标记位

既然垃圾回收算法要求给对象加上垃圾回收的标记，显然是需要有标记位的。一般会在对象中加上一个标记域，或者利用对象指针的低位进行标记，而 Go 语言的做法与 C 语言的结构体对象完全一致，都是非侵入式的标记位。

Go 把标记放在了堆区 arena 前面对应的 bitmap 里，要知道获得对象区域的类型信息是实现精确的垃圾回收的前提。这个 bitmap 位图区域的每个字（不是 byte，而是 word，大小是 32b 或 64b）对应 4 位的标记位。因此，64 位系统中相当于每个标记 bitmap 的 B 对应 16 个堆中的 B。

仅供非商业用途或交流学习使用




```
// 获取 b 对应的 bitmap 位图
obj, hbits, span, objIndex := heapBitsForObject(obj, b, i)
mbits := span.markBitsForIndex(objIndex)
// 判断是否被标记过，已标记或不是指针都不入队
mbits.isMarked()
hbits.hasPointers(span.elemsize)
```

如上所示，heapBitsForAddr 可以获取对应堆地址的 bitmap 位 hbits，根据它可以判断是否是指针，如果是指针且之前没有被标记过，则标记当前对象为可达，并且 greayObject 入队，供给其他的 markWorker 来处理。

3. 清理触发

启动垃圾回收的时机也是一个难题，因为如果频繁垃圾回收会浪费 CPU，影响应用代码的性能，而如果垃圾回收启动太晚，会导致堆内存累计太多，如何平衡这个问题是 Go 语言的每个版本更新时都要面对的。

每一次 mallocgc 都会检查是否需要 gcstart，触发条件由两个参数决定：gc_trigger 和 gcpercent。

gc_trigger 初始为 4MB，next_gc 初始为 4MB，之后每次标记完成时将重新计算动态调整值大小。但 gc_trigger 至少要大于初始的 4MB，同时至少要比当前使用的 heap 大 1MB 才会触发 GC 操作。

这个检查是在堆上分配大于 32KB 对象的时候进行，此时检查是否满足垃圾回收条件，如果满足则进行垃圾回收。

```
func mallocgc(size uintptr, typ *_type, needzero bool) unsafe.Pointer {
    ...
    shouldhelpgc := false
    // 分配的对象小于 32KB
    if size <= maxSmallSize {
        ...
    } else {
        shouldhelpgc = true
        ...
    }
    ...
    // gcShouldStart() 函数进行触发条件检测
    // gcBackgroudMode 表示并发 gc
    if shouldhelpgc && gcShouldStart(false) {
        gcStart(gcBackgroundMode, false)
    }
}
```

仅供非商业用途或交流学习使用




```
func gcShouldStart(forceTrigger bool) bool {
      return gcphase == _GCoff && (forceTrigger ||
            memstats.heap_live >= memstats.gc_trigger) &&
       memstats.enablegc && panicking == 0 &&
       gcpercent >= 0
}

func gcinit() {
    _ = setGCPercent(readgogc())
    memstats.gc_trigger = heapminimum
    memstats.next_gc = uint64(float64(memstats.gc_trigger) / (1 +
      gcController.triggerRatio) * (1 + float64(gcpercent)/100))
    work.startSema = 1
    work.markDoneSema = 1
}
func gcMark() {
    memstats.gc_trigger = uint64(float64(memstats.heap_marked) *
       (1 + gcController.triggerRatio))
}
```

上面是自动垃圾回收，还有一种是主动垃圾回收，通过调用 runtime.GC()，这是阻塞式的。有意思的是，gcmark 在每次标记结束后会重置阈值大小。如果当前使用了 4MB 内存，这时设置 gc_trigger 为 2×4MB，也就是当内存分配到 8MB 时会再次触发 GC；回收之后内存为 5MB，那下一次就要达到 10MB 才会触发 GC。这个比例（triggerRatio）是由 gcpercent/100 决定的。

gcpercent 的值是通过环境变量 GOGC 获取的，如果不设置这个环境变量，默认值是 100。如果将它设置成 off，则关闭垃圾回收。

如果系统启动或短时间内大量分配对象，会将垃圾回收的 gc_trigger 推高。在服务正常后，活跃对象远小于这个阈值，造成垃圾回收无法触发，这个问题交给 sysmon 解决，它每隔 2 分钟强制触发 GC 一次。这个 forcegc 的 goroutine 一直驻留在后台，直到 sysmon 将它唤醒开始执行 GC 而不用检查阈值。

15.4.2 标记实现

GC 开始之前，有一些准备工作，整个 GC 启动过程都是 STW 状态，它启动了所有将并发执行标记工作的 goroutine，然后进入 GCMark 状态启动写屏障，启动 gcController：

```
func gcStart(mode gcMode, forceTrigger bool) {
    ...
    // 在后台启动 mark worker
    if mode == gcBackgroundMode {
```

仅供非商业用途或交流学习使用




```
        gcBgMarkStartWorkers()
    }
    ...
    // Stop The World (Go 源码关键注释语句，表示此时所有的任务都会被暂停)
    systemstack(stopTheWorldWithSema)
    // 完成之前的清理工作
    systemstack(func() {
        finishsweep_m()
    })
      ....
      // 进入 Mark 状态，写屏障
    if mode == gcBackgroundMode {
        // GC 开始前的准备工作
            gcController.startCycle()
        // 处理设置 GCPhase，setGCPhase 还会开启“写屏障”
        setGCPhase(_GCmark)

        gcBgMarkPrepare() // 必须在辅助启用之前发生
        gcMarkRootPrepare()

      // 标记所有活跃的 tinyalloc 内存块
        gcMarkTinyAllocs()

        // 恢复整个世界（线程恢复）
        systemstack(startTheWorldWithSema)
    } else {
        ...
    }
}
```

gcstart 会为所有的 P 都准备好对应的 goroutine worker，但是这些 worker 需要被 gcController 的 findRunnableGCWorker 唤醒才能工作。M 启动后会一直通过 schedule 查找可执行的 G，其中 gcworker 也是 G 的一部分，但是首先要检查当前状态是不是回收状态，如果是才唤醒 worker 开始标记工作。

标记阶段是并行的，通过在后台一直运行标记 worker 来实现：

```
func gcBgMarkStartWorkers() {
     // 后台并发标记由每个 P 的 G 执行，确保每个 P 都有一个后台 GC 的 G 在运行
    for _, p := range &allp {
        if p == nil || p.status == _Pdead {
            break
        }
        if p.gcBgMarkWorker == 0 {
            go gcBgMarkWorker(p)
            notetsleepg(&work.bgMarkReady, -1)
            noteclear(&work.bgMarkReady)
```

仅供非商业用途或交流学习使用




```
        }
    }
}
// gcBgMarkWorker 是一直在后台运行的，大部分时候是休眠状态，通过 gcController 来调度
func gcBgMarkWorker(_p_ *p) {
    for {
        // 将当前 goroutine 休眠，直到满足某些条件
        gopark(...)
        ...
        // mark 过程
        systemstack(func() {
        // 标记 gortoutine 为可占用，然后它所在堆栈才会被扫描
        // 这样，两个标记的 worker 相互扫描，从而避免死锁
        // 由于不能修改 G 栈上的任何内容，因此从 G 堆栈读取是安全的
        casgstatus(gp, _Grunning, _Gwaiting)
        switch _p_.gcMarkWorkerMode {
        default:
            throw("gcBgMarkWorker: unexpected gcMarkWorkerMode")
        case gcMarkWorkerDedicatedMode:
            gcDrain(&_p_.gcw, gcDrainNoBlock|gcDrainFlushBgCredit)
        case gcMarkWorkerFractionalMode:
            gcDrain(&_p_.gcw, gcDrainUntilPreempt|gcDrainFlushBgCredit)
        case gcMarkWorkerIdleMode:
            gcDrain(&_p_.gcw, gcDrainIdle|gcDrainUntilPreempt|gcDrainFlushBgCredit)
        }
        casgstatus(gp, _Gwaiting, _Grunning)
        })
        ...
    }
}
```

标记代码主要在函数 gcDrain()中实现：

```
/* gcDrain 扫描工作缓冲区中的 root 区和对象，使灰色的对象变黑，直到所有的 root 区域和工作缓
冲区域都被排空 */
func gcDrain(gcw *gcWork, flags gcDrainFlags) {
    ...
    // 处理 root 区域标记
    if work.markrootNext < work.markrootJobs {
        for !(preemptible && gp.preempt) {
            job := atomic.Xadd(&work.markrootNext, +1) - 1
            if job >= work.markrootJobs {
                break
            }
            markroot(gcw, job)
            if idle && pollWork() {
                goto done
            }
        }
```

仅供非商业用途或交流学习使用




```
        }

        // 处理 heap 标记
        for !(preemptible && gp.preempt) {
            ...
            // 从灰色列队中取出对象
            var b uintptr
            if blocking {
                b = gcw.get()
            } else {
                b = gcw.tryGetFast()
                if b == 0 {
                    b = gcw.tryGet()
                }
            }
            if b == 0 {
                // 工作遇到障碍或 tryGet 失败
                break
            }
            // 扫描灰色对象的引用对象，标记为灰色，入灰色队列
            scanobject(b, gcw)
        }
    }
```

结束后调用 gcMarkDone，这里会引起 StopTheWorld，接下来进入 gcMarkTermination 中的 gcMark 阶段。

15.4.3 清理

清理相对来说就简单很多了。有多个地方可以触发 sweep，比如 GC 标记结束会触发 gcSweep。如果是并发清除，需要回收从 gc_trigger 到当前活跃内存大小相同的 heap 区域，唤醒后台的 sweep goroutine。

```
func gcSweep(mode gcMode) {
    ...
    // 阻塞式
    if !_ConcurrentSweep || mode == gcForceBlockMode {
        // 在特殊情况下，同步扫描
        ...
        // 快速扫描所有的 span
        for sweepone() != ^uintptr(0) {
            sweep.npausesweep++
        }
        // 多做一次 mProf_GC，因为所有“空闲”的事件此刻都是真实存在的
      mProf_GC()
        mProf_GC()
```

仅供非商业用途或交流学习使用




```
        return
    }

    // 并行式，后台扫描
    lock(&sweep.lock)
    if sweep.parked {
        sweep.parked = false
        ready(sweep.g, 0, true)
    }
    unlock(&sweep.lock)
}

// 在 runtime 初始化时进行 gcenable
func gcenable() {
    go bgsweep(c)
}
func bgsweep(c chan int) {
    sweep.g = getg()

    lock(&sweep.lock)
    sweep.parked = true
    c <- 1
    goparkunlock(&sweep.lock, "GC sweep wait", traceEvGoBlock, 1)

    for {
        for gosweepone() != ^uintptr(0) {
            sweep.nbgsweep++
            Gosched()
        }
        lock(&sweep.lock)
        if !gosweepdone() {
            // 如果 GC 在 gosweepone 返回^0 之前并在锁之后运行，则可能发生这种情况
            unlock(&sweep.lock)
            continue
        }
        sweep.parked = true
        goparkunlock(&sweep.lock, "GC sweep wait", traceEvGoBlock, 1)
    }
}

func gosweepone() uintptr {
    var ret uintptr
    systemstack(func() {
        ret = sweepone()
    })
    return ret
}
```

仅供非商业用途或交流学习使用



对于并行式清理，在 GC 初始化的时候就会启动 bgsweep()，然后在后台一直循环，它会执行 gosweepone。sweepone（一个内置的检查方法）首先会遍历所有的 spans 看它的 sweepgen 是否需要检查，如果要就检查这个 MSpan 里所有的 object 的 bit（位），看是否需要回收。这个过程可能触发 MSpan 到 MCentral 的回收，最终可能回收到 MHeap 的空闲列表当中。在空闲列表当中的内存在超过一定阈值时间后会被 sysmon 建议交还给内核。

15.4.4 监控

上面讲到有两种 GC 触发方式，分别是自动检测和用户主动调用。除此之外 Go 语言本身还会对运行状态进行监控，如果超过两分钟没有 GC，则触发 GC。监控函数就是 sysmon()，在主 goroutine 中启动。

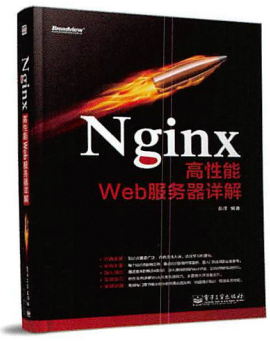
```
// 不管有没有P，一直运行，所以不允许写障碍
func sysmon() {
    ...
    for {
        now := nanotime()
        unixnow := unixnanotime()

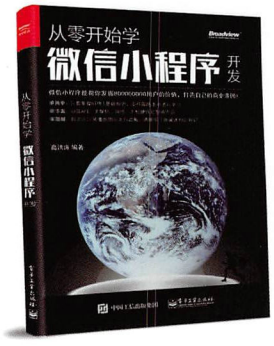
        lastgc := int64(atomic.Load64(&memstats.last_gc))
        if gcphase == _GCoff && lastgc != 0 && unixnow-lastgc > forcegcperiod && ato
mic.Load(&forcegc.idle) != 0 {
            lock(&forcegc.lock)
            forcegc.idle = 0
            forcegc.g.schedlink = 0
            injectglist(forcegc.g)//将 forcegc goroutine 加入 runnable queue
            unlock(&forcegc.lock)
        }
    }
}
```

15.5 本章小结

本章主要讲解了 Go 语言的内存分配原理以及具体实现过程。虽然 Go 语言的垃圾回收还有一些不足的地方需要改进，但是可以看到每次更新 Go 语言的版本时，垃圾回收的部分都有一些改进。有理由相信 Go 语言在庞大社区驱动以及 Google 的支持下会越来越好。

相关好书





Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

Go语言核心知识点

入门
- 程序结构
- 数据类型
- 流程控制

并发编程
- 协程
- 通道
- 同步

基础知识
- 函数
- 复合类型
- 切片与映射

网络编程
- Socket
- HTTP
- RPC
- 数据库

包与结构体
- 包
- 结构体
- 方法

标准库
- 输入 / 输出
- 数据结构
- 文件系统
- 文件处理
- 日志记录

接口与反射
- 接口
- 反射

内存管理
- 源码剖析
- 实现机制
- 代码优化

博文视点Broadview
新浪微博 weibo.com
@博文视点Broadview

策划编辑：张月萍
责任编辑：牛　勇
封面设计：吴海燕

上架建议：计算机\网络编程

ISBN 978-7-121-34966-9
9 787121 349669 >
定价：79.00元